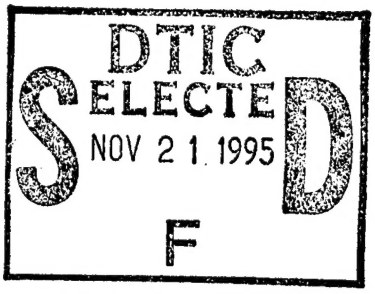
ADVANCED NETWORKING AND DISTRIBUTED SYSTEMS
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY

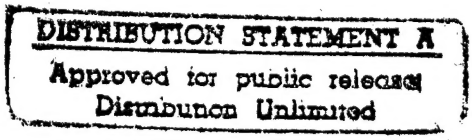
ANNUAL TECHNICAL REPORT

June 1, 1993

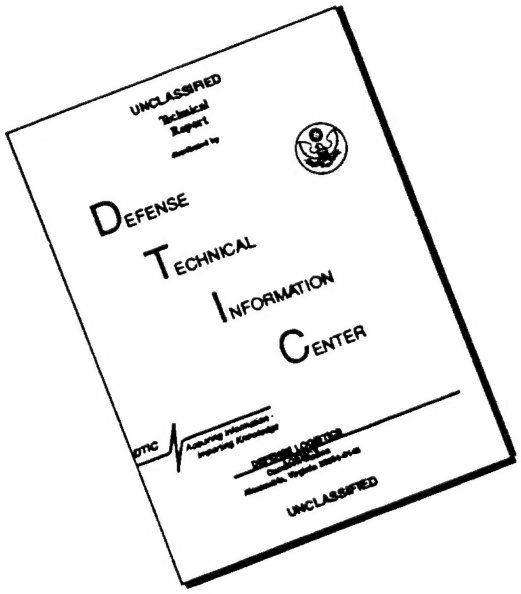


Principal Investigator: Leonard Kleinrock

Computer Science Department
School of Engineering and Applied Science
University of California
Los Angeles



DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

UNABLE TO GET MISSING PAGES FROM ABSTRACT 2,

4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, & 38

PER PATRICK RODELL(703) 696-0094 AT DEFENSE

ADVANCED RESEARCH PROJECT AGENCY

ARLINGTON, VA.

MAY 22, 1996

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Advanced Networking & Distributed Systems Annual Technical Report		5. TYPE OF REPORT & PERIOD COVERED Annual Technical Report June 1, 1992 - June 30, 1993
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Leonard Kleinrock		8. CONTRACT OR GRANT NUMBER(s) MDA 972-91-J-1011
9. PERFORMING ORGANIZATION NAME AND ADDRESS School of Engineering & Applied Science University of California, Los Angeles Los Angeles, CA 90024-1596		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS DARPA Order No. 7728 DSRS-37204
11. CONTROLLING OFFICE NAME AND ADDRESS ARPA/CSTO 3701 N. Fairfax Drive Arlington, VA 22203-1714		12. REPORT DATE
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for Public Release: Distribution Unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)		

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ADVANCED SYSTEMS LABORATORY

ANNUAL TECHNICAL REPORT

June 1, 1993

Contract Number: MDA 972-91-J-1011
DARPA Order Number: 7728
Contract Period: June 1, 1991 to May 31, 1994
Report Period: June 1, 1992 to May 31, 1993

Principal Investigator: Leonard Kleinrock

(310) 825-2543

Computer Science Department
School of Engineering and Applied Science
University of California, Los Angeles

Sponsored by

DEFENSE ADVANCED RESEARCH PROJECTS AGENCY

ADVANCED NETWORKING AND DISTRIBUTED SYSTEMS

Defense Advanced Research Projects Agency

Annual Technical Report

May 30, 1993

This Yearly Technical Report covers research carried out on the Advanced Networking and Distributed Systems Contract at UCLA under DARPA Contract Number MDA 972-91-J-1011 covering the period from June 1, 1991 through May 30, 1992. Under this contract we have the following statement of work comprising five tasks:

STATEMENT OF WORK

Topic A: High Speed Networking

Task A1: Fast Packet Switching Using Multistage Interconnection Networks

We propose to investigate the performance of a variety of Multistage Interconnection Networks such as the Starlite network. We will develop analytical models to evaluate the throughput and response time of the overall traffic in the case of uniform traffic as well as certain forms of hot spot traffic. We will also evaluate the behavior of Message Combining to eliminate the effects of hot spots. A transformation and superposition method is being developed to be used with the analytical model to evaluate any given general traffic pattern (e.g., multiple hot spots). A delay model analysis comparing the discarding switch and the blocking switch will also be developed. We also propose to study a structured buffered pool scheme to prevent normal traffic from being blocked by the saturated tree caused by hot spot traffic.

Task A2: Analysis Of Competing Lightwave Networks

The use of Wavelength Division Multiple Access (WDMA) optical switching for high-speed packet networks is a predictable development in the evolution of fast packet switching. We propose to evaluate the behavior of single-hop WDMA optical switching, using agile receiver filters. Whereas our main thrust will be on these single-hop structures, we will also look at multi-hop access using fixed filters. We will compare the response time, blocking and throughput for each.

TOPIC B: ARCHITECTURE AND PARALLEL PROCESSING

Task B1: Performance Of Boolean n-Cube Interconnection Networks

We propose to evaluate the performance of Boolean n-cube interconnection networks for parallel processing systems. The focus will be on data communication issues rather than on processing issues. By exploiting the homogeneity property of Boolean n-cube interconnection networks, we can design non-blocking routing algorithms with limited size buffers. A technique called referral is used to guarantee that every node accepts all the messages transmitted from its neighbors. This type of routing algorithm is critical in any implementation. Store-and-forward is one such routing algorithm. In this scheme, time is divided into cycles to which the network is synchronized. In each cycle every node simultaneously transmits some of its stored messages to its neighbors. An analytical model will be developed to predict the network performance under different traffic patterns. We also intend to design an intelligent routing algorithm to improve the performance. Another routing scheme to consider is a modified version of virtual cut-through. Virtual cut-through is a scheme such that when a message arrives at an intermediate node and its selected outgoing channel is free, then the message is sent to the adjacent node before it is completely received at this intermediate node. Therefore, the delay due to unnecessary buffering in front of an idle channel is avoided. Modified virtual cut-through is also a non-blocking algorithm. We will investigate the (positive or negative) effect of adding additional buffers to a node in this case. We are further interested in non-uniform traffic problems in Boolean n-cube networks.

We also propose to study the performance of these networks in a hostile and/or unreliable environment. In this environment, nodes and links may disappear and also unreliable (i.e., noisy) transmissions may occur.

Task B2: Distributed Simulation

Parallel asynchronous simulation methods (such as Time Warp) offer an optimistic alternative to synchronous conservative approaches to distributed simulation. We propose to evaluate the speedup of P processors conducting a parallel asynchronous simulation using analytic and simulation tools. We already have an exact solution for the case of two processors (P=2). Also, we have upper bounds on the best one can do by letting the P processors run ahead of each other as compared to forcing them to synchronize at every step. We are interested in extending the results to P processors and to include the effect of queued messages. Furthermore, we propose to investigate the use of the linear Poisson process as a model for these systems.

Task B3: A New Model Of Load Sharing

We are interested in studying the behavior of interacting processes which gobble up processing resources in their neighborhood. In particular, if we begin with a one-dimensional world, we can place processes on a ring, where there is a quantity of processing power distributed uniformly around the ring. A process requires a changing amount of processing capacity. As its needs increase, the process attempts to grow in both directions along the ring until it either has enough

capacity, or it bumps into another process moving in its direction, in which case they both stop moving toward each other. As time progresses, a process may or may not have all the capacity it needs. The object is to study the response time of jobs represented by such processes in a limited resource, competitive environment. Clearly, this model extends to higher dimensions, and we propose to study the case where processors are distributed over a multi-dimensional hypersphere. The effect of distributed load sharing in this environment will be evaluated.

Accomplishments for this Period

During we have graduated 1 Ph.D. student, have had 9 papers published, 2 papers accepted and 2 papers submitted to the professional literature. Progress continues to be made in our stated goals.

We have encountered no obstacles to our research and have made some major contributions in new areas as well as in extensions of previous work on this grant.

We have developed some important results in the area of parallel processing systems. In particular, we established a metric which allows us to model the behavior of programs in a way which identifies the optimum number of processors one should assign to a job. This has led to a generalization of Amdahl's Law as well as a simple rule that specifies the optimum number of processors. One paper, "On Parallel Processing Systems: Amdahl's Law Generalized and Some Results on Optimal Design" by Kleinrock and Huang, which is attached to this report, addresses the optimization problem in the case where any processors that go idle in the course of the computation are not allowed to be temporarily reassigned to other jobs waiting in the queue. In another paper (also attached), "Performance Evaluation of Dynamic Sharing of Processors in Two-Stage Parallel Processing Systems" by Huang and Kleinrock does allow this reuse of idle processors by waiting jobs.

We had earlier reported on our results for TIme-Warp (optimistic) Simulation methods. A related study, represented by the attached paper entitled "The Virtual Time Data-Parallel Machine" by Shen and Kleinrock, evaluated the performance of aggressive processing of ready tasks at the instruction level. This approach shows significant increases in performance over the synchronized approach to instruction execution.

A universal result was established in the attached paper, "Depth- First Heuristic Search on a SIMD Machine" by Powley, Ferguson and Korf. This result determined the optimum times when a SIMD processing system should do load balancing.

In many of our investigations, we continue to encounter the issue of how to use processing power that has temporarily become available. We address the gains to be had in this case directly in the attached paper, "Collecting Unused Processing Capacity: An Analysis of Transient Distributed Systems" by Kleinrock and Korfhage. We determine the response time of a job when it has available a randomly changing number of processors that cooperate in its execution.

Our work in high speed networking with fiber optics continues to produce valuable results. Indeed, we have established the performance limits of an ideal fiber-optic wave-length-division switch as a function of the number of tunable and fixed transmitters and receivers. This work is reported in the attached paper, "Performance Analysis of Single-Hop Wavelength Division Multiple Access Networks" by Lu ad Kleinrock.

Below we list our publications for this period. Progress continues along a number of fronts, and we are beginning to study the performance of multi-hop, multi-channel wireless communication systems in a mobile environment, as well as adaptive and self-regulating automata in a loosely coupled environment.

ANDS PUBLICATIONS
DARPA CONTRACT - MDA 972-91-J-1011

Computer Science Department
University of California, Los Angeles
Professor Leonard Kleinrock

June 1, 1992 to May 31, 1993

Ph.D DISSERTATIONS

1. Lu, Jonathan Chunhsien, "Design and Analysis of Wavelength Division Multiple Access Lightwave Packet Networks, June 1992.

PAPERS PUBLISHED IN PROFESSIONAL AND SCHOLARLY JOURNALS

Lu, J. and L. Kleinrock, "A Wavelength Division Multiple Access Protocol for High-speed Local Area Networks with a Passive Star Topology," *Performance Evaluation, Vol. 16, No.1-3, pp. 223-239, November 1992.*

Kleinrock, L. and R. Felderman, "Two Processor Time Warp Analysis: A Unifying Approach," *International Journal in Computer Simulation,* Volume 2, Number 4, pp. 345-371, 1992, UCLA Technical Report CSD-920034.

Huang, J.H. and L. Kleinrock, "Performance Evaluation of Dynamic Sharing of Processors in Two-Stage Parallel Processing Systems," *IEEE Transactions on Parallel and Distributed Systems,* Vol.4, No.3, March 1993, pp.306-317.

Powley, C., C. Ferguson and R. Korf, "Depth-First Heuristic Search on a SIMD Machine," *AI Journal,* April 1993, pp.199-242.

Kleinrock, L., "On the Modeling and Analysis of Computer Networks", Special Issue of IEEE Proceedings, March 2, 1993.

Huang, J.H. and L. Kleinrock, "Throughput Analysis and Protocol Design for CSMA and BTMA Protocols under Noisy Environment," *IEE Proceedings-I, Vol. 139, No. 3, pp.289-296, June 1992*

Lin, T-I. and Kleinrock, L., "Performance Analysis of Finite-Buffered Multistage Interconnection Networks with Alternative Switching Architectures", IFIP Workshop TC6, La Martinique, French Carribean Island, January 1992.

Shen, S. and L. Kleinrock, "The Virtual Time Data-Parallel Machine" *4th Symposium on the*

Frontiers of Massively Parallel Computation, IEEE Computer Society, McLean, VA, October 1992, pp 46-53..

Kleinrock, L. and W. Korfhage, "Collecting Unused Processing Capacity: An Analysis of Transient Distributed Systems", IEEE TPDS, May 1993, pp. 535,546.

PAPERS ACCEPTED FOR PUBLICATION

1. Tung, Brian and L. Kleinrock, "Distributed Control Methods" accepted to *2nd International Symposium on High Performance Distributed Computing,* Spokane Washington, July 21,23, 1993.

2. Felderman, R. and L. Kleinrock, "Two Processor Time Warp Analysis: Capturing the Effects of Message Queueing and Rollback/State Saving Costs," UCLA Technical Report CSD-920035, accepted for Memorial Issue for Felix Pollaczek. AEU special issue "Teletraffic Theory and Engineering".

PAPERS SUBMITTED FOR PUBLICATION

1. Lu, J. C. and Kleinrock, L., "A WDMA Protocol for Multichannel DQDB Networks" submitted to GLOBECOM '93, January 1993

2. Tung, Brian and L. Kleinrock "Distributed Control Using Simple Automata" submitted to *IEEE Transactions on Parallel and Distributed Systems,* December 1992

PAPERS IN PREPARATION

1. Kleinrock, L. and D. Nielsen, "Data Structures and Algorithms for the Efficient Implementation of Structural Level Reduction of the GSPN Model", to be submitted to *International Workshop on Petri Nets and Performance Models.*

2. Kleinrock, L. and D. Nielsen, "Analysis Techniques to Increase the Computational Power of the GSPN-Reward Model", to be submitted to *International Workshop on Petri Nets and Performance Models.*

THE VIRTUAL-TIME DATA-PARALLEL MACHINE

Shioupyn Shen
Leonard Kleinrock

Reprinted from PROCEEDINGS OF THE FOURTH SYMPOSIUM ON THE FRONTIERS OF MASSIVELY PARALLEL COMPUTATION (FRONTIERS '92), McLean, Virginia, October 19-21, 1992

IEEE Computer Society
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1264

Washington, DC • Los Alamitos • Brussels • Tokyo

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

IEEE COMPUTER SOCIETY

The Virtual-Time Data-Parallel Machine

Shioupyn Shen and Leonard Kleinrock

Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90024-1596

Abstract

We propose the "Virtual-Time Data-Parallel Machine" to execute SIMD (Single Instruction Multiple Data) programs *asynchronously*. We first illustrate how asynchronous execution is more efficient than synchronous execution. For a simple model, we show that asynchronous execution outperforms synchronous execution roughly by a factor of $(\ln N)$, where N is the number of processors in the system. We then explore how to execute SIMD programs asynchronously without violating the SIMD semantics. We design a *FIFO priority cache*, one for each processing element, to record the recent history of all variables. The cache, which is stacked between the processor and the memory, supports asynchronous execution in hardware efficiently *and* preserves the SIMD semantics of the software transparently. Analysis and simulation results indicate that the Virtual-Time Data-Parallel Machine can achieve *linear speed-up* for computation intensive data-parallel programs when the number of processors is large.

1 Introduction

For the past twenty years, solid state technology has been much more successful in reducing the cost of VLSI chips than in increasing the peak speed of ECL circuits. As a direct result, the architectural superiority of supercomputers is vanishing because we can easily implement most of the advanced features of supercomputers on a single chip[1]. The performance gap between the fastest processor (in terms of MIPS) and the most cost-effective processor (in terms of MIPS/$) is diminishing rapidly. In the future, the key to supercomputing will not be the high speed of a single processor; instead, it will be the high degree of parallelism.

The difficulties of parallel processing are two-fold. The first problem is that the computational model is hard to use (for asynchronous execution) and the second problem is that the hardware efficiency is poor (for synchronous execution). We propose the "Virtual-Time Data-Parallel Machine" to solve *both* problems at once. The concept of this machine is derived from "The Connection Machine" [4] and "Virtual Time" [6].

The Connection Machine introduced the data-parallel computational model [5]. The SIMD semantics of the *data-parallel* model make it easy to develop parallel programs and make it capable of expressing fine-grain parallelism. Though the Connection Machine achieves significant speed-up for a large number of processors, hardware efficiency may be poor because of its requirement for synchronous execution.

Virtual Time introduced the "Time Warp" synchronization mechanism for parallel discrete event simulation [7]. The *optimistic* approach of Time Warp eliminates unnecessary blocking, and therefore makes better use of the hardware. However, it is hard to generalize Virtual Time to other paradigms of parallel processing.

We suggest the use of Time Warp to execute data-parallel programs asynchronously in hopes of exploiting more parallelism and obtaining better efficiency. By performance modeling, we show that the efficiency (i.e., the sustained speed over the raw speed) of the system is asymptotically 40% for a large number of processors; this is a significant improvement over the traditional approach.

The organization of this paper is as follows: Section 2 provides the motivation, Section 3 explores the key concept, Section 4 addresses performance modeling, Section 5 describes the hardware support, Section 6 discusses the extensions, and the last section concludes the paper.

2 Motivation – The Interconnection Network Bottleneck

The data-parallel approach has been very successful in solving or avoiding many of the technical difficulties of parallel processing.

Data-parallel computers would be the obvious champion in the parallel processing arena if the interconnection network were not the bottleneck.

The performance of the data-parallel approach is more sensitive to the interconnection network than that of the other approaches because of its SIMD semantics. Even though all processors start executing the same instruction simultaneously, they seldom finish this instruction together for many reasons. For

[1] There are approximately three million transistors in an Intel 80586 microprocessor but only two million transistors in a CRAY-1 supercomputer.

0-8186-2772-7/92 $3.00 © 1992 IEEE

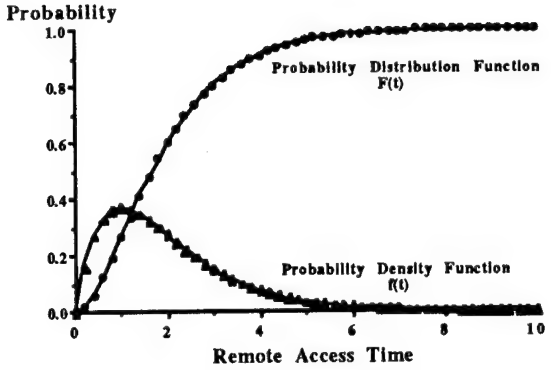


Figure 1: A hypothetical example of remote access time distribution.

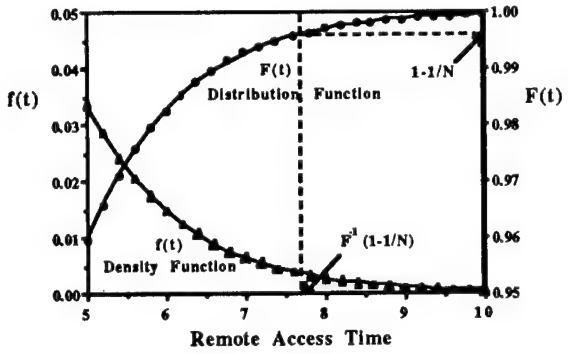


Figure 2: The longest remote access time for N processors is $F^{-1}(1-\frac{1}{N})$, which is mainly determined by the long tiny tail of the probability density function $f(t)$.

example, the access of remote operands (i.e., variables on other processing elements) may take vastly different amounts of time due to network contention and blocking. As a result, the execution time of an instruction varies among the processors.

Figure 1 shows a hypothetical example of the remote access time distribution, where $f(t)$ and $F(t)$ are the probability density function and probability distribution function of the remote access time, respectively. Though the remote access time distributions for various interconnection networks are different, they have several characteristics in common – they have large mean and variance, and more importantly, their probability density functions have long, tiny tails. The long tiny tail has little influence on the mean remote access time because it is so tiny. However, it is the long (though tiny) tail that drives the performance down.

The synchronous execution of SIMD programs forces a processor which finishes the instruction early wait until all processors finish this instruction. Therefore, what really counts is the longest execution time (in other words, the worst case in remote access time) across all processors. The maximum value of N independent samples is approximately $F^{-1}(1-\frac{1}{N})$, where $F^{-1}(t)$ is the inverse function of $F(t)$. For large N, the above term is determined by the long, tiny tail of $f(t)$ as shown in Fig. 2.

In our experience and that of the others, the critical bottleneck of data-parallel computing is the interconnection network. Even though the bandwidth of the interconnection network is large, we cannot justify the cost of providing sufficient bandwidth to reduce the maximum remote access time for random communication patterns. We would like to know how good the performance will be if we smooth out the variation of the remote access time. If the performance is very good, we also would like to figure out how to do it at acceptable cost.

First, we show that the system can achieve linear speed-up (constant efficiency) for a large number of processors. Second, we propose a very cost-effective solution (asynchronous execution) to reduce the susceptibility to the variation of remote access time without modifying the interconnection network. Our approach is to attach minimal hardware support to every processing element to "neutralize" the network hazards, instead of resorting to an expensive "upgrade" of the interconnection network. We can achieve roughly the same performance as if the remote access time is constant but with *twice* the mean. This approach trades the variance for a larger mean. The above trade-off is favorable because the major bottleneck is in the variance instead of the mean, especially for systems with a large number of processors.

In addition to the remote operand fetch, the actual computation of the instruction sometimes introduces large variations into the instruction execution time as well. Conditional enabling/disabling is a common practice in data-parallel programming. Even though it takes constant amount of time for the enabled processors to execute the instruction, collectively speaking, the instruction execution time varies because it takes no time for the disabled processors to skip the instruction. If the disabling probability is high[2], then the execution time varies a lot. In this paper, we use the generic term "instruction execution time", which may refer to either the remote access time or the computation time or both.

[2]For example, a tree-reduction operation [5] of size N takes $(\log_2 N)$ iterations for a total of $(N-1)$ operations. The disabling probability is as high as $(1-\frac{1}{\log_2 N})$.

3 The Key Concept – Asynchronous SIMD

The Connection Machine is a typical example of the traditional data-parallel machine, which has the following characteristics – (i) SIMD, (ii) distributed memory, (iii) massive parallelism, and (iv) programmable interconnection. We are perfectly happy with these properties except the first one – SIMD, or more precisely, synchronous execution of SIMD programs. The inefficiency of synchronous execution comes from unnecessary blocking. Processors that finish the current instruction early are blocked until all processors finish this instruction, even though the operands needed to execute the next instruction may be available. As we know, synchronous execution is a direct way to enforce the "SIMD semantics", but what really matters is the SIMD semantics itself, rather than the synchronous execution.

SIMD semantics is in fact a kind of causality constraint, which is explained as follows. The execution of the i-th instruction (an event "scheduled" at "simulation" time i) depends on the execution of the (i-1)-th instruction (an event scheduled at simulation time i-1). The sequence count of the instruction stream is analogous to the simulation time, which specifies when an event *should* happen, in contrast to "real-time" when the event *does* happen. The data-dependency constraint of the SIMD semantics is thus equivalent to the causality constraint of parallel discrete event simulation (PDES).

The synchronous execution of SIMD programs is essentially the time-stepped execution of PDES, which is considered an inefficient implementation of PDES. On the other hand, the optimistic approach of Virtual Time employs periodic state-saving so that processors have more freedom to go ahead instead of being blocked unnecessarily. The Virtual-Time Data-Parallel Machine takes a similar (but not identical) approach – the execution of the next instruction can proceed independently of the progress on other processors as long as its own data dependencies are satisfied and its current state is properly saved.

Figure 3 illustrates how asynchronous execution of SIMD programs is far more efficient than the conventional synchronous execution. In a task graph, nodes correspond to tasks (instructions) and links correspond to causality constraints (data dependencies). Figure 3.a shows the intrinsic data dependencies of an example program, which ignores all artifacts due to the execution model. When synchronous execution is enforced, it is equivalent to adding more links to the task graph such that every task depends on all the tasks one row above it. Figure 3.b shows the large number of additional data dependencies of the program caused by the requirements of the synchronous execution model.

We know that adding/removing links to a task graph decreases/increases the parallelism of the task graph, respectively. The Virtual-Time Data-Parallel Machine promotes asynchronous execution by removing those extra links associated with synchronous execution (i.e., to achieve better performance) while preserving the original data dependencies (i.e., without sacrificing the semantics).

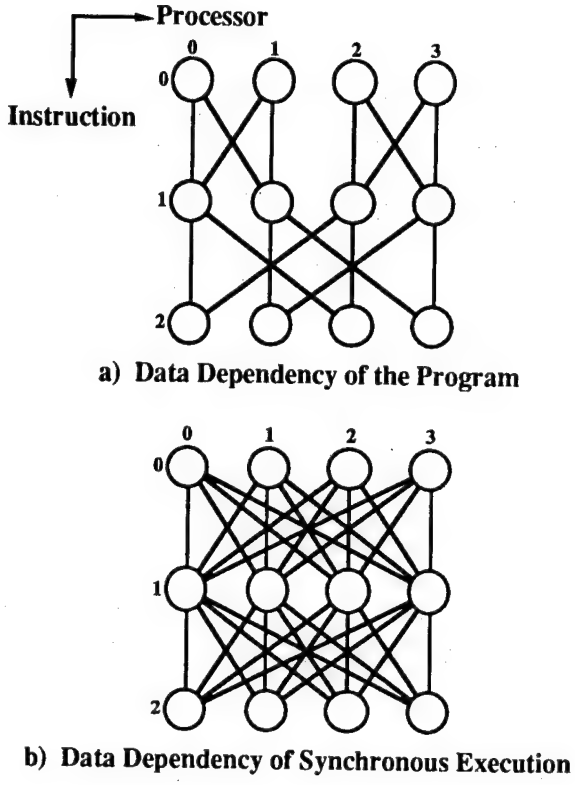


Figure 3: The task graph representation of data dependencies.

4 Performance Modeling – Partial Synchronization

A simple model of the Virtual-Time Data-Parallel Machine is the "partial synchronization" model [1], in contrast to "total synchronization" model (i.e., barrier synchronization). The model is as follows. The SIMD machine consists of N homogeneous processors, i.e., every processor has the same processing power and executes the same instruction stream such that the behavior of every processor is *statistically* equivalent. The task graph (Fig. 4) is an ∞ by N matrix, one column for each processor. Each task depends on a set of tasks one row above it. Let the size of this set be A, which is the number of immediate ancestors of this task. If $A = N$ for all tasks, then it is total synchronization; otherwise, it is partial synchronization. We are interested in the case where A is a small number[3].

[3]If the tasks correspond to instructions, then the value of A is analogous to the number of operands of an assembly instruction, which is usually small.

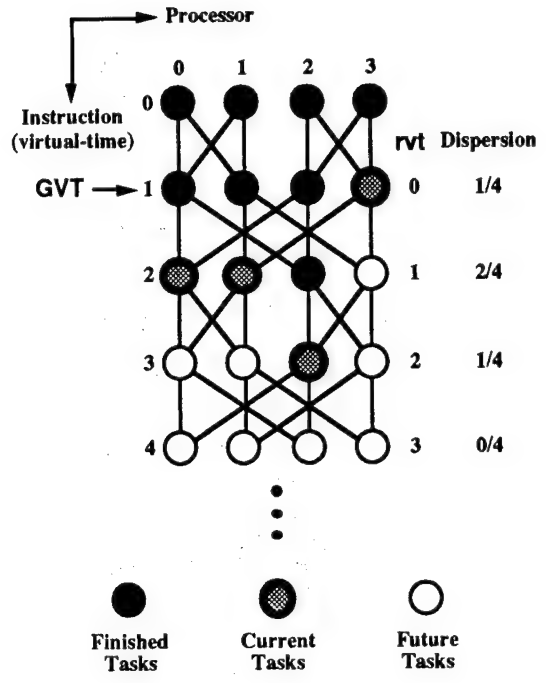


Figure 4: A snapshot of a task graph in execution.

For partial synchronization, the instructions in execution spread out in virtual-time[4]. However, this dispersion is confined due to the data dependency constraints. In order to describe the dynamic behavior of the system, we introduce the following terms (refer to Fig. 4). The *global virtual-time* (GVT) of the system is defined to be the minimum virtual-time of all processors, i.e.,

$$\texttt{GVT} \triangleq \min_{\forall proc.} \{\text{virtual-time}\} \tag{1}$$

The *relative virtual-time* (rvt) of an instruction (or a processor) is defined to be the difference between its virtual-time and the GVT, i.e.,

$$\texttt{rvt} \triangleq \text{virtual-time} - \texttt{GVT} \tag{2}$$

The dispersion function describes how processors scatter in virtual-time, i.e.,

$$\texttt{dispersion}(i) \triangleq \text{Prob}[\texttt{rvt} = i] \tag{3}$$

which can be interpreted as the probability that the rvt of a (tagged) processor is i (averaged over a long period of real-time), or as the distribution of processors in virtual-time (at some real-time instant). Figure 4 shows a snapshot of the system, and Figure 5 illustrates its dynamic behavior.

[4]In this paper, "virtual-time" and "simulation-time" are used interchangeably, as are "instruction" and "task".

We make the following assumptions for the purpose of performance modeling:

1. The execution time of tasks is exponentially distributed.
2. The number of immediate ancestors (A) is exactly two.
3. One ancestor is the preceding task on the same processor, and the other ancestor is uniformly distributed among the tasks in the preceding row.

Though these assumptions are not particularly realistic, they are simple enough to capture some fundamental insight as to how asynchronous execution outperforms synchronous execution.

We are interested in the following performance measures:

Speed-Up:

$$\text{Speed-Up} \triangleq \frac{\text{execution time with a } \textit{single} \text{ processor}}{\text{execution time with } N \text{ processors}} \tag{4}$$

Efficiency:

$$\text{Efficiency} = \frac{\text{Speed-Up}}{N} \tag{5}$$

Efficiency-Gain:

$$\text{Efficiency Gain} = \frac{\text{efficiency of } \textit{asynchronous} \text{ execution}}{\text{efficiency of } \textit{synchronous} \text{ execution}} \tag{6}$$

Figure 6 shows the speed-up and efficiency of *synchronous* execution from analysis [2]. The efficiency of synchronous execution drops (to zero) as the number of processors increases. Figure 7 shows the speed-up and efficiency of *asynchronous* execution as obtained from simulation[5]. Analytical results [8] show that the asymptotic efficiency for a large number of processors is approximately 40%. Figure 8 shows that the efficiency gain is proportional to the logarithm of the number of processors. From this figure we see the motivation for considering asynchronous execution.

We now address the *scalability* of the Virtual-Time Data-Parallel Machine. The traditional definition of scalability is with respect to the speed-up of running the *same* program on an increasing number of processors. This definition is not directly applicable to data-parallel machines where the number of processors is on the same order as the intrinsic parallelism of the program. When the number of processors increases, the problem size must increase *proportionally* so that the intrinsic parallelism increases proportionally as well. If a system scales up well, the execution time is relatively constant.

[5]Analyses are also available [8] for an upper-bound, a lower bound, and an approximation to the efficiency of asynchronous execution.

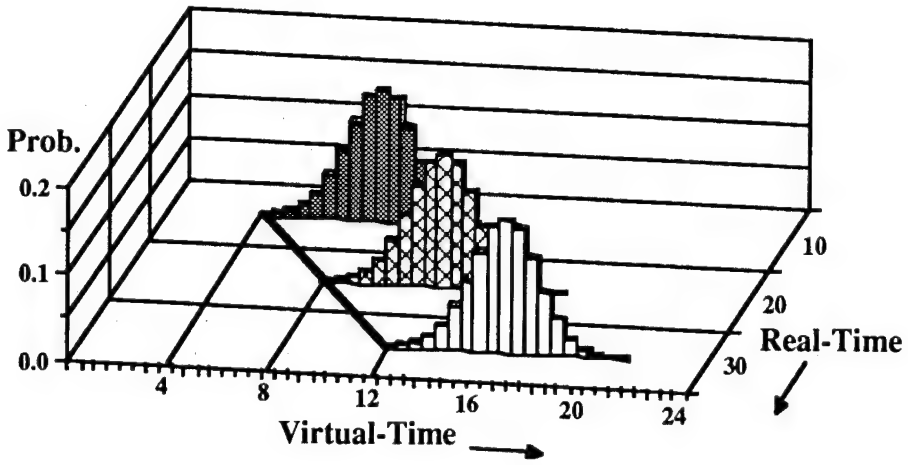


Figure 5: The distribution of processors in virtual-time at several real-time instants. $GVT(t)$, global virtual-time as a function of real-time, summarizes the progress of program execution at real-time t. In this figure, $GVT(10) = 4$, $GVT(20) = 8$, and $GVT(30) = 12$.

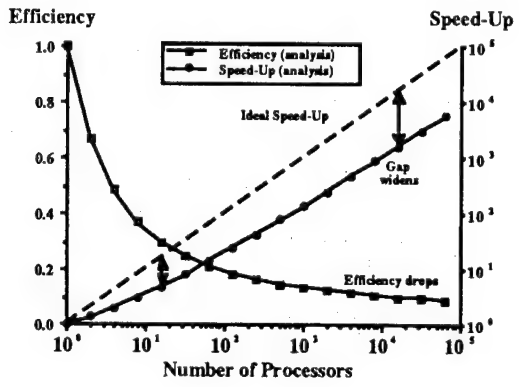


Figure 6: Speed-up and efficiency of synchronous execution.

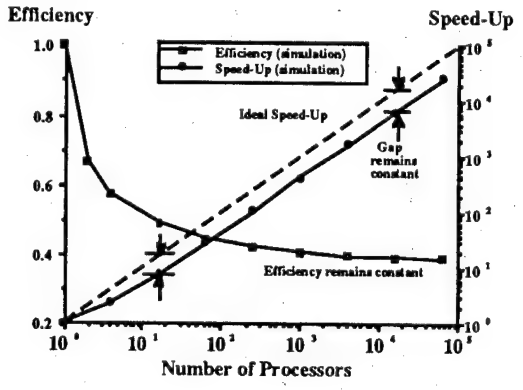


Figure 7: Speed-up and efficiency of asynchronous execution.

An architecture simulator has been developed to run "real" data-parallel programs on the Virtual-Time Data-Parallel Machine to illustrate its superb scalability. The main assumption[6] made in the simulator is the randomness in instruction execution time (in fact, the remote access time). Figure 9 shows the time required to solve a Laplace's equation asynchronously versus synchronously. This diagram reveals that asynchronous execution scales up well as shown by the almost constant execution time, while the execution time for synchronous execution increases. The above argument does *not* mean that asynchronous execution is more favorable than synchronous execution in solving partial differential equations since, had the instruction execution time been constant, as if, with such equations, then the synchronous and asynchronous approaches would have behaved similarly.. However, it indicates that asynchronous execution is capable of smoothing out the variations of instruction execution time.

[6] We also make other assumptions on the number of cycles for integer and floating-point operations. As long as the number of cycles for computation is less than that for communication, the simulation results are not sensitive to these assumptions.

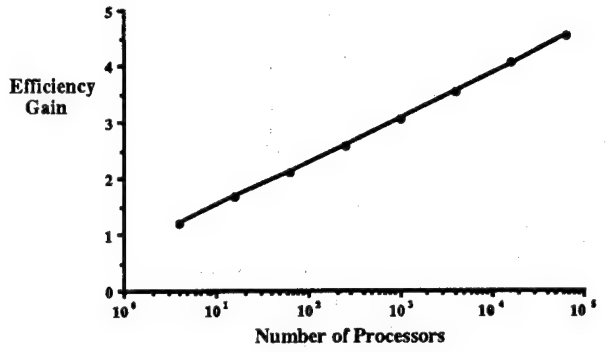


Figure 8: The efficiency gain – asynchronous over synchronous.

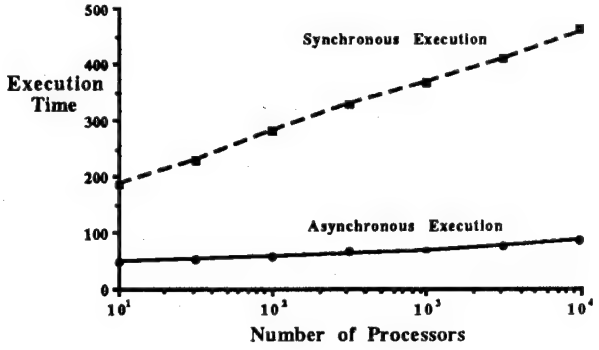


Figure 9: The execution time of a data-parallel program when the number of processors increases with the problem size.

5 Hardware Support – The FIFO Priority Cache

During asynchronous execution, processors are allowed to spread out at different virtual-times. Consider the case in which one processor at a smaller virtual-time (say, i) sends a memory request[7] to another processor at a larger virtual-time (j, where $j > i$). The time (virtual-time) of the request is *current* to the former processor but *previous* to the latter processor. The requested value has actually been generated in a previous instruction (before i) on the latter processor, and is subject to being overwritten by instructions between i and (j-1). In order to prevent overwriting useful data, every processor must save all previous values of its variables (i.e., the memory "history") back to GVT(since no processor has a virtual-time earlier than GVT, no values earlier than GVTwill be requested in the future). For practical reasons, there is a physical limit on the size of the memory history, say K. If a processor goes so fast that it is K instructions ahead of GVT, it must be temporarily suspended because it has used up its memory history. Analysis [8] shows that if K is greater than ($\ln N$), the above situation rarely occurs and the performance hardly degrades due to the limited size of memory history.[8]

Memory history is so important and so frequently used that it deserves special hardware support. We have designed a "FIFO priority cache" (which implements the *incremental backup* algorithm) for the memory history (one cache per processor) with the following characteristics.

FIFO Queueing: Write requests are not executed immediately; instead, they are pushed into a FIFO queue (of size K). If the queue is full, the oldest pending write request (i.e., the one with the earliest virtual-time) is popped out of the FIFO queue and then sent to the main memory (i.e., to update the main memory).

Associative Search: For every read request whose virtual-time is less than or equal to the program counter of the processor, we conduct an *associative* search (like a cache memory) in the queue for hits, where a hit is any pending write request in the queue with matching address and earlier virtual-time.

Priority Arbitration: If there is more than one hit, we choose the latest hit, i.e., the one with the largest virtual-time. Priority arbitration (e.g., choosing the latest hit) can be implemented by a priority encoder/decoder pair. If there is no hit (i.e., a cache miss), then we consult the main memory because the requested value is stored in the main memory.

Related research on the space-time memory can be found in [3] and [8].

Synchronous SIMD machines can hardly benefit from the cache memory because a cache miss for one processor is aggravated to a cache miss for the whole system. Asynchronous execution allows the system to take full advantage of the cache memory technology to resolve the speed discrepancy between the CPU and the main memory. A small FIFO priority cache not only supports the memory history, but also accelerates memory references.

6 Extensions – Two-Phase Write

Even though we can smooth out the variations of the remote access time, the interconnection network may

[7] Memory requests are time-stamped to unambiguously specify the requested values.

[8] What a coincidence! ($\ln N$) happens to be the performance gain as well.

still be the performance bottleneck (however, the bottleneck is less severe than before). The problem resides in the mismatch of *local* access time and *remote* access time. The technology is such that the speed of the processor (and the memory) has improved quickly but the speed (in terms of latency instead of throughput) of the interconnection network has improved at a much slower rate. As a result, the remote access time is tens to one hundred times larger than the local access time, and the mismatch will become even worse in the future. The processors spend most of their time waiting for remote operands because the actual computation or the local references can be done in a flash. If we could pipeline the instruction execution, the *throughput* of the system would improve dramatically. However, the variation of remote access time makes pipelining difficult if not impossible.

Traditional pipelining is "synchronous" pipelining in the sense that timing information is known in advance so that a reservation table synchronizes the resource allocation. Data-flow is "asynchronous" pipelining in the sense that only the data-dependency counts and timing information is either unknown or irrelevant. Since this paper is promoting the *asynchronity*, whenever the "synchronous" approach fails, try its "asynchronous" counterpart.

The sequential semantics of SIMD programs adds an implicit ancestor to every instruction on every processor – the preceding instruction on each of these processors. However, we observe that the result of an instruction may not be used immediately by the next instruction. If the current instruction is blocked (e.g., waiting for a remote operand), the execution of the next instruction can proceed without waiting for the current instruction to finish. Before the next instruction starts executing, the processor must schedule the execution of the current instruction and invalidate the variables that may be modified by the current instruction. Such a problem was addressed many years ago by the Tomasulo algorithm [9]. This algorithm, used by the floating point unit of the IBM 360/91, converts sequential computation into data-flow computation within a small sliding window of instructions.

The main idea is to separate a `write` operation of a variable into two phases – the *logical* write and the *physical* write. The logical write is executed first before the content of the write operation is available; it invalidates the variable and assigns a unique identifier to the content of the write operation. From then on, all read requests to that variable (before the variable is overwritten) are transformed to waiting for that identifier. The next instruction can proceed after the logical write without waiting for the physical write. The physical write is executed when the content of the variable becomes available; it is sent to every processor waiting for the corresponding identifier. Once we adopt the two-phase write, then head-of-the-line blocking, which enforces sequential execution, is eliminated; at the same time, the sequential semantics are preserved. Thus we see that the techniques used to compensate for the long pipeline stages of floating point arithmetic units in sequential machine may now be used to compensate for the long (network) delays due to remote access in parallel processing systems.

The two-phase write can be easily implemented in the memory history by adding an extra *busy* bit to every outstanding update. A logical write sets the busy bit to one, representing the fact that an update is taking place, and the content is not available. A physical write resets the busy bit to zero, representing the fact that the data in this update is available. References to a busy update receive the time-stamped address of the update (which serves as the unique identifier), and then get blocked. When a physical write is executed, all references to the matching identifier are unblocked.

With the two-phase write, the Virtual-Time Data-Parallel Machine converts the SIMD computation from control-flow to data-flow (within a small sliding window of neighboring instructions). Data-flow execution *recovers* more threads of execution than control-flow, which increases the concurrency and improves the efficiency of the Virtual-Time Data-Parallel Machine.

7 Conclusions

Long and unpredictable remote access latency is often the performance bottleneck of massively parallel computing. Asynchronous execution of the Virtual-Time Data-Parallel Machine provides one way to relieve this bottleneck. We have proposed some minimal modifications to the architecture of the traditional data-parallel machine (e.g., the CM-2), which converts the way it executes SIMD programs from synchronous to asynchronous. We have provided a basic foundation for the understanding of both *why* and *how* to improve the efficiency of SIMD programs by allowing asynchronous execution.

Asynchronous execution makes the machine more MIMD (Multiple Instruction Multiple Data)-like. It is nevertheless a SIMD machine from the programmer's point of view! A more detailed discussion of these ideas and a comprehensive quantitative analysis of this machine can be found in [8].

Acknowledgments

This research has been supported by the Advanced Research Projects Agency of the Department of Defense under contract MDA 903-87-C0663, Parallel Systems Laboratory.

References

[1] C. S. Chang and R. Nelson, "Bounds on the Speedup and Efficiency of Partial Synchronization in Parallel Processing Systems," Research Report RC 16474, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, January 1991.

[2] Robert E. Felderman and Leonard Kleinrock, "An Upper Bound on the Improvement of Asynchronous versus Synchronous Distributed Processing," *Proceedings of the SCS Multiconference on Distributed Simulation*, Vol. 22, No. 1, pp. 131–136, January 1990.

[3] Richard M. Fujimoto, "The Virtual Time Machine," *International Symposium on Parallel Algorithms and Architectures*, pp. 199–208, June 1989.

[4] W. Daniel Hillis, *The Connection Machine*, The MIT Press, Cambridge, Massachusetts, 1985.

[5] W. Daniel Hillis and Guy L. Steele, Jr., "Data Parallel Algorithms," *Communications of the ACM*, Vol. 29, No. 12, pp. 1170–1183, December 1986.

[6] David R. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, pp. 404–425, July 1985.

[7] Jayadev Misra, "Distributed Discrete-Event Simulation," *ACM Computing Surveys*, Vol. 18, No. 1, pp. 39–65, March 1986.

[8] Shioupyn Shen, *The Virtual-Time Data-Parallel Machine*, Ph. D. Dissertation, Computer Science Department, UCLA, 1991.

[9] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal*, pp. 25–33, January 1967.

Performance Evaluation of Dynamic Sharing of Processors in Two-Stage Parallel Processing Systems

Jau-Hsiung Huang, *Member, IEEE*, and Leonard Kleinrock, *Fellow, IEEE*

***Abstract*—In this paper, we first study the performance of job scheduling in a large parallel processing system where a job is modeled as a concatenation of two stages which must be processed in sequence. We denote Pi as the number of processors required by stage i and denote P as the total number of processors in the system. The service time requirement of stage i, given the required Pi processors, is exponentially distributed with mean $1/\mu_i$. Hence, a job can be fully described by a quadruple (P_1, P_2, μ_1, μ_2). Three service disciplines which can fully utilize all processors in the system are studied in this paper.**

We first consider a *large* parallel computing system where $\text{Max}(P_1, P_2) \geq P >> 1$ and $\text{Max}(P_1, P_2) >> \text{Min}(P_1, P_2)$. For such systems, exact expressions for the mean system delay are obtained for various job models and disciplines. Our results show that the priority should be given to jobs working on the stage which requires fewer processors. We then relax the *large* parallel system (i.e., $P >> 1$) condition to obtain the mean system time for two job models when the priority is given to the second stage. Moreover, a *Scale-up Rule* is introduced to obtain the approximated delay performance when the system provides more processors than the maximum number of processors required by both stages (i.e., $P > \text{Max}(P_1, P_2)$). Lastly, an approximation model is given for jobs with more than two stages.

***Index Terms*—High-concurrency stage, low-concurrency stage, parallel processing system, processor sharing, Scale-up rule.**

I. Introduction

In this paper, we first consider a parallel computing system in which jobs are composed of two stages which must be processed in sequence. In each stage, up to a given maximum number of processors can be used concurrently and the number of processors required by different stages need not be the same. For instance, we can view a job as a program and a stage in a job as a procedure in the program where each procedure can be executed using a certain amount of processors concurrently. The number of processors required by a procedure depends on how many processors it can use in its algorithm. We denote P_i as the number of processors required by stage i and we denote P to be the total number of processors in the system. The service time of stage i given the required P_i processors is exponentially distributed with mean $1/\mu_i$. Hence, a job can be fully described by a quadruple (P_1, P_2, μ_1, μ_2). A general concept of this kind of job model was first proposed in [11].

Manuscript received July 19, 1990; revised June 3, 1991 and September 14, 1991. This work was supported by the Defense Advanced Research Projects Agency, Department of Defense, under Contracts MDA 903-87-C-0064 and MDA 903-87-0663.

J.-H. Huang is with the Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan.

L. Kleinrock is with the Computer Science Department, University of California, Los Angeles, Los Angeles, CA 90024.

IEEE Log Number 9205852.

For easier notation, we denote the *low-concurrency stage* as the stage where the number of processors required equals $\text{Min}(P_1, P_2)$. Similarly, we denote the *high-concurrency stage* as the stage where the number of processors required equals $\text{Max}(P_1, P_2)$. Hence, the job models considered in this paper may be either 1) the low-concurrency stage precedes the high-concurrency stage or 2) the high-concurrency stage precedes the low-concurrency stage. These two job models are denoted as the *LH (Low-High) job model* and *HL (High-Low) job model*, respectively.

Other than considering the characteristics of jobs, we propose three service disciplines for such systems. The basic principle of the service disciplines studied in this paper is to fully utilize all processors such that we do not allow the system to have idle processors while there are jobs waiting in the queue. To achieve this, the system has to share processors among jobs according to a service discipline. Two of the disciplines studied in this paper use priority schemes which assign the priority to a job according to which stage the job is working on. The third discipline follows a straight First Come First Serve rule. Hence the disciplines studied are 1) priority given to jobs working on stage 1 with preemption, 2) priority given to jobs working on stage 2 with preemption, and 3) FCFS without preemption. More details about job models and service disciplines are given in the following section.

For the 2-stage job model, we first find the mean system time of LH and HL job models under different kinds of disciplines. From the obtained mean system time analysis, we then find the best service discipline to minimize the mean system time for both LH and HL job models. These results shed light on designing the operating system for a parallel computing system.

The paper is organized as follows. In Section II, we give a detailed description to the job models and various service disciplines. In Section III, we assume the number of processors required by the low-concurrency stage is far fewer than that of the high-concurrency stage (i.e., $\text{Max}(P_1, P_2) >> \text{Min}(P_1, P_2)$) and we further assume the number of processors in the system is no more than the number of processors required by the high-concurrency stage (i.e., $P \leq \text{Max}(P_1, P_2)$). The performance of various combinations of job models and service disciplines are given and a comparison to find the best service discipline is also provided. In Section IV, we drop the $\text{Max}(P_1, P_2) >> \text{Min}(P_1, P_2)$ condition and find the delay performance using the discipline which gives the priority to jobs working on stage 2 with preemption. In Section V, we propose a *Scale-up Rule* which gives a good approximation

result of the mean system time when the number of processors in the system is more than the number of processors required by the high-concurrency stage (i.e., $P > \text{Max}(P_1, P_2)$). Section VI contains an approximation model for jobs with more than two stages. The concluding remarks are given in Section VII.

II. Model Description and Assumptions

In our model, jobs arrive to the system according to a Poisson process with rate λ. Whenever a job in a stage requires more processors than the system currently can provide, this job simply uses all the processors available to it with an appropriately elongated stage service time such that the work done for that stage remains unchanged. That is, the service time for that stage will still be exponentially distributed but with a larger mean. For example, if a job in a stage can use 10 processors for one second of work and if there are only 5 processors available, it will use these 5 processors and require two seconds of work to complete its work. This elongated stage service time with 5 processors ensures the conservation of the work required in that stage. As mentioned earlier, this is to fully utilize all the processors in the system. An example of an LH job model is shown in Fig. 1(a) and an HL job model is shown in Fig. 1(b) where $\text{Max}(P_1, P_2)/\text{Min}(P_1, P_2)$ equals m.

The first two service disciplines considered in this paper are a version of priority queueing with preemption. If the total number of processors occupied by higher priority jobs in the system is less than P, those processors which are not needed by these higher priority jobs are assigned to lower priority jobs. The assignment of priorities to a job is based upon which stage the job is currently working on. Hence, when a job finishes the first stage and advances to the second stage, the priority ranking of this job will be changed.

For instance, if the priority is assigned to jobs working on stage 2, then a job advances from stage 1 to stage 2 will gain a higher priority. Moreover, in cases where $P_2 > P_1$ and jobs working in stage 2 have higher priorities, a job advances from stage 1 to stage 2 will achieve a higher priority and will need more processors to work on stage 2. In this case, this job will preempt processors from those jobs working on stage 1 until either it has gained enough processors for stage 2 or there are no more jobs working on stage 1. Those jobs with all processors preempted will be pushed back to the head of the queue waiting for available processors. The third discipline considered in this paper does not allow processor preemption and follows a First Come First Serve rule.

Furthermore, for all three disciplines, if there are more than one job wanting to share processors in the same stage, we will first satisfy the processor requirement of the first job before allocating processors to the second job and so on. This process continues until all processors are allocated to jobs or until all jobs are satisfied. If there are jobs which do not receive processors, they stay in the queue of that stage. Hence, the service discipline for each stage is a FCFS scheme for all disciplines. The details of the service disciplines follows.

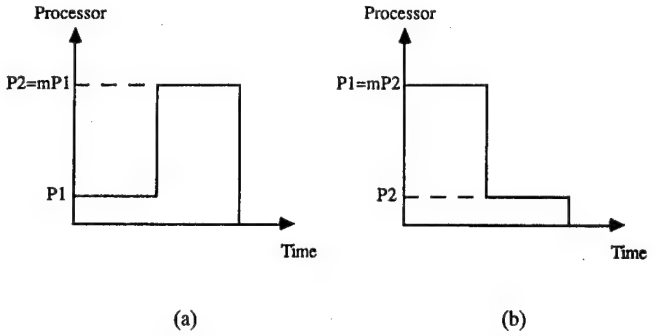


Fig. 1. LH and HL job models. (a) LH job model. (b) HL job model.

A. Priority Given to Jobs Working on Stage One with Preemption

A job is allowed to receive service as long as there are available processors in the system following a preemptive priority service discipline. A job's priority decreases when it finishes stage 1 and enters stage 2. Under such a preemptive priority scheme, when a job enters the system while there are no available processors in the system, it can preempt processors from jobs working on stage 2, if there is any, to start servicing its stage 1.

B. Priority Given to Jobs Working on Stage Two with Preemption

In this discipline, a job's priority increases when it finishes stage 1 and enters stage 2. Hence, when a job enters stage 2 and requires more processors than it currently possesses from stage 1, it can preempt processors from jobs working on stage 1, if there is any, to start servicing its stage 2.

C. FCFS Discipline without Preemption

The discipline introduced here is a FCFS discipline where preemption is not allowed. A job is allowed to receive service as long as there are available processors in the system. Further, the processors occupied by a job cannot be preempted by any other jobs. However, if there are processors released by a job, those jobs already in service which need more processors have a higher priority than jobs in the queue to occupy the released processors. A job may release processors by either advancing from the high-concurrency stage to the low-concurrency stage or by finishing stage 2 and leaving the system.

III. Performance Evaluation of Different Job Models Under Various Disciplines

In this section, we consider a large parallel computing system where $\text{Max}(P_1, P_2) \geq P >> 1$ and $\text{Max}(P_1, P_2) >> \text{Min}(P_1, P_2)$. Before further discussion, we will first examine the cases when $P < \text{Max}(P_1, P_2)$. When $P < \text{Max}(P_1, P_2)$, the high-concurrency stage can actually use only P processors, hence the quadruple job description should be modified. For instance, if stage 2 is the high-concurrency stage (i.e., $P_2 > P > P_1$), then the quadruple (P_1, P_2, μ_1, μ_2) should be modified as $(P_1, P, \mu_1, P/P_2\mu_2)$. That is, stage 2 can use all P processors for an elongated service time. Hence, we will only consider the case when $P = \text{Max}(P_1, P_2)$ without loss of generality.

To give an example for the job model discussed in this section, the low-concurrency stage can be regarded as a *serial* stage which requires only one processor and the high-concurrency stage as a *parallel* stage which can use all processors in the system. For such a system, job models shown in Fig. 1(a) and (b) can be approximated as shown in Fig. 2(a) and (b), respectively since the number of processors required by the low-concurrency stage is negligible compared to that required by the high-concurrency stage. For the rest of this section, we will use these approximated job models for analysis. It will be shown in Section IV-A and Fig. 6 and when $\text{Max}(P_1, P_2) \geq 20\text{Min}(P_1, P_2)$, the delay performance of the job models shown in Fig. 1 is very close to the job models shown in Fig. 2, which will be analyzed in this section.

For such a system, the low-concurrency stage requires a negligible amount of total processors while the high-concurrency stage requires *all* P processors in the system. Therefore, if the high-concurrency stage has a higher priority, then there can be at most one job working on the high-concurrency stage since it will occupy all processors in the system. Under such a circumstance, all other jobs in the system will be forced to wait in the queue since there are no available processors left.

On the other hand, if the low-concurrency stage has a higher priority, then all jobs working on the low-concurrency stage can work concurrently and they will have no effect on jobs working on the high-concurrency stage because the processors taken by jobs in the low-concurrency stage are negligible. That is, in this case, there can be as many jobs working on the low-concurrency stage and one job working on the high-concurrency stage at the same time.

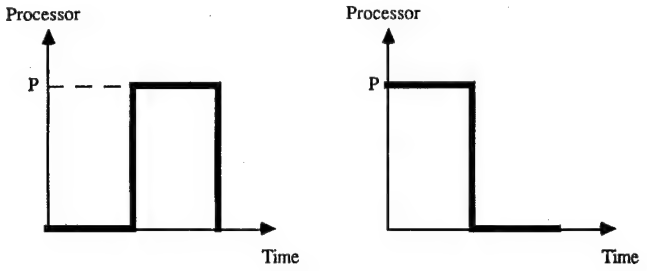
In this section, we will give results of the mean system time of LH and HL job models under various disciplines. Clearly, there will be six models from a combination of two job models and three scheduling disciplines. These six models are classified into three groups in according to the service disciplines. Note that the formula of infinite server queues are approximations to the actual system.

A. Priority Given to Jobs Working on Stage One with Preemption

In this section, we give the priority to jobs working on stage 1. For the two job models, we denote T_{LH} and T_{HL} as the mean system time for the LH job model and the HL job model, respectively. These notation will be used throughout this section.

Theorem 1: For systems with priority given to jobs working on stage 1 with preemption, the Laplace Transform, $Y^*_{HL}(s)$, of the system time of the HL job model is shown at the bottom of this page where ρ_1 equals λ/μ_1 and $G^*(s)$ is the root of the following quadratic equation:

$$G^*(s)^2 - \frac{s+\lambda+\mu_1}{\lambda} G^*(s) + \frac{\mu_1}{\lambda} = 0.$$



(a) (b)

Fig. 2. The approximated LH and HL job models when $P = \text{Max}(P_1, P_2) >> \text{Min}(P_1, P_2)$. (a) LH job model. (b) HL job model.

Proof: See Appendix A.

Theorem 2: For systems with priority given to jobs working on stage 1 with preemption,

$$T_{HL} = \frac{1/\mu_1 + 1/\mu_2}{1-\rho_1} + \frac{\lambda}{\mu_1^2(1-\rho_1)^2} \tag{2}$$

$$T_{LH} = \frac{1}{\mu_1} + \frac{1/\mu_2}{1-\lambda/\mu_2}. \tag{3}$$

Proof: Equation (2) can easily be derived from (1). To find T_{LH}, since stage 1 has higher priority, stage 1 can be regarded as an $\text{M}/\text{M}/\infty$ system. Further, since jobs in stage 1 occupy no processors at all, stage 2 can be regarded as an $\text{M}/\text{M}/1$ queue. Hence, T_{LH} can be shown as in (3). Q.E.D.

Note that $\lambda/\mu_1 (= \rho_1)$ is the system load for the HL job model since only stage 1 will contribute workload to the system. Similarly, $\lambda/\mu_2 (= \rho_2)$ is the system load for the LH job model. These can be seen from (2) and (3).

B. Priority Given to Jobs Working on Stage Two with Preemption

For systems which give priorities to jobs working on stage 2 with preemption, we first derive the z-transform of the number of jobs in the system as given in the following equation.

$$P(z) = \frac{\mu_2 - \lambda}{\mu_2 - \lambda z} \cdot \exp\left(\lambda/\mu_1 \cdot \left[z - 1 + \ln \frac{\mu_2 - \lambda}{|\lambda z - \mu_2|}\right]\right) \tag{4}$$

where $P(z)$ is defined as the z-transform of the number of jobs in the system. The proof of (4) is provided in Appendix B. From (4), we obtain the following theorem.

Theorem 3: For systems with priority given to stage 2 with preemption,

$$T_{LH} = \frac{1/\mu_1 + 1/\mu_2}{1-\lambda/\mu_2} \tag{5}$$

$$T_{HL} = \frac{1/\mu_1}{1-\lambda/\mu_1} + \frac{1}{\mu_2}. \tag{6}$$

$$Y^*_{HL}(s) = \frac{\mu_1^2\mu_2(1-\rho_1)^2}{G^*(s)[s+\mu_1(1-\rho_1)][S+\lambda-\lambda G^*(s)+\mu_2][s-\lambda G^*(s)+\mu_1]} \tag{1}$$

Proof: From (4), we can easily derive the mean number of customers in the system for the LH job model and we can hence prove (5) using Little's result [12]. Again, to find T_{HL} is easy since stage 1 performs like an M/M/1 queue and stage 2 is like an M/M/∞ queue. Q.E.D.

C. First Come First Serve Discipline

The performance of First Come First Serve discipline for the HL job model is the same as when the priority is given to stage 2 with preemption and the reason follows. Since jobs in stage 1 possess more processors than they will need in stage 2, hence whenever a job finishes stage 1 and enters stage 2, it does not need to ask for more processors than it already possesses. Actually it will release processors to other jobs. Hence, all jobs in stage 2 will always have enough processors. Therefore, the FCFS discipline for the HL job model is the same as the case when the priority is given to stage 2.

However, for the LH job model, the exact result is not obtained. Nonetheless, we can easily show that the performance is worse than the case when priority is given to stage 1 with preemption by the following reason. For FCFS discipline, whenever there is a job in stage 2, new arriving jobs cannot start receiving service for the low-concurrency stage since all processors are occupied by old jobs in stage 2. Hence, these new jobs would have to wait in the queue before receiving service for the low-concurrency stage. However, if the discipline used is to give priority to stage 1, then these new arriving jobs can immediately start receiving service for the low-concurrency stage and will not have to wait in the queue. Moreover, jobs in stage 1 do not really occupy processors; hence, it does not interfere with jobs working on stage 2. Therefore, the mean system time for the case when priority is given to stage 1 with preemption for the LH job model is better than that using the FCFS nonpreemptive service discipline. Applying the similar argument, we can show that the LH job model performs better than the case when priority is given to stage 2 with preemption. From these conclusions, we arrive at the following results for the FCFS system.

$$T_{HL} = \frac{1/\mu_1}{1-\lambda/\mu_1} + \frac{1}{\mu_2} \tag{7}$$

$$\frac{1/\mu_1 + 1/\mu_2}{1-\lambda/\mu_2} > T_{LH} > \frac{1}{\mu_1} + \frac{1/\mu_2}{1-\lambda/\mu_2}. \tag{8}$$

D. Performance Comparison Among Various Disciplines

Comparing (3), (5), and (8), we find that for LH job models, the system which gives priority to stage 1 with preemption has the smallest mean system time. On the other hand, for HL job models, we find that the system which gives priority to stage 2 with preemption and FCFS without preemption achieve the smallest average system time by comparing (2), (6), and (7). From the results derived above, we arrive at the following conclusions:

1) FCFS nonpreemptive discipline is not the optimal discipline.
2) The best service discipline depends on job models.
3) For the disciplines considered in this paper, the priority should be given to the low-concurrency stage which requires fewer processors.

Conclusion 3) above may not seem obvious. As a known fact, the best service discipline to minimize the mean system time is to give higher priority to jobs with less remaining service time if that is known. In our 2-stage job model, jobs working on stage 2 clearly are closer to completion than jobs working on stage 1; hence, the priority should be given to jobs working on stage 2 for both job models. However, from conclusion 3) stated above, we know that the priority should be given to stage 1 in the LH job model. This is opposite to our intuitive reasoning.

The reasons for this can be explained as follows. The problem of giving the priority to the high-concurrency stage is that by so doing, whenever there is a job in the high-concurrency stage, all jobs in the low-concurrency stage will be forced to wait since all processors are occupied by the job in the high-concurrency stage. Once all jobs in the high-concurrency stage finish services, all the processors will be released to jobs waiting to receive service for the low-concurrency stage. Since the number of processors required by all jobs working in the low-concurrency stage is small compared to the number of processors in the system, therefore, most of the processors will be idle. It is this waiting time in the low-concurrency stage and the inefficiency of utilizing processors which causes the poor performance.

On the other hand, if the priority is given to the low-concurrency stage, then all jobs in the low-concurrency stage would not have to wait in any case. Further, jobs in the high-concurrency stage can also receive service since the number of processors occupied by jobs in the low-concurrency stage is negligible. Hence, by giving the priority to the low-concurrency stage obtains a better delay performance for all job models. An example is shown in Fig. 3 for $(P_1, P_2, \mu_1, \mu_2) = (1, 20, 1, 1)$ and $P = 20$. Fig. 3 shows that the system with priority given to jobs working on stage 1 has the best performance while the performance of the system with FCFS discipline is very close to it. In Fig. 3, the mean system time for the FCFS discipline is obtained by simulations.

This conclusion can be further extended to a 3-stage job model where a job is composed of a low-concurrency stage followed by a high-concurrency stage followed by another low-concurrency stage as shown in Fig. 4. From the results obtained in Theorems 2 and 3 and by defining ρ_2 to be λ/μ_2, we can easily obtain the mean system delay for the following cases.

Case 1: If the highest priority is given to jobs working on stage 3, the next priority to jobs working on stage 2, and the lowest priority to jobs working on stage 1; we have

$$T_{LHL} = \frac{1/\mu_1 + 1/\mu_2}{1-\rho_2} + 1/\mu_3.$$

Case 2: If the highest priority is given to jobs working on stage 3, the next priority to jobs working on stage 1, and the lowest priority to jobs working on stage 2; we have

$$T_{LHL} = 1/\mu_1 + \frac{1/\mu_2}{1-\rho_2} + 1/\mu_3.$$

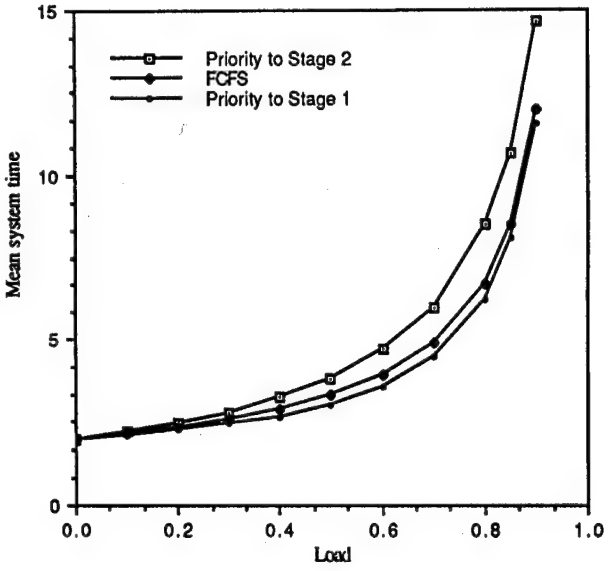


Fig. 3. The comparison for three service disciplines for $(P_1, P_2, \mu_1, \mu_2) = (1, 20, 1, 1)$ and $P = 20$.

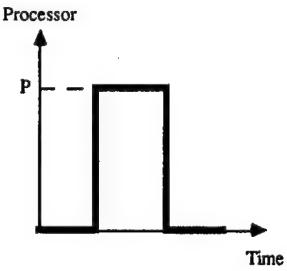


Fig. 4. A 3-stage job model with one high-concurrency stage and two low-concurrency stages.

Case 3: If the highest priority is given to jobs working on stage 1, the next priority to jobs working on stage 2, and the lowest priority to jobs working on stage 3; we have

$$T_{LHL} = 1/\mu_1 + \frac{1/\mu_2 + 1/\mu_3}{1-\rho_2} + \frac{\lambda}{\mu_2^2(1-\rho_2)^2}$$

Note that in Case 1, we give the priority to jobs closer to completion while in Case 2 the priority is given to jobs working on stages which require fewer processors. From these results, we can clearly see that Case 2 has the smallest mean system time. Hence, our conclusion in this section also works for 3-stage job models with one high-concurrency stage.

IV. Delay Analysis by Relaxing $\mathrm{Max}(P_1, P_2) >> \mathrm{Min}(P_1, P_2)$ Condition

In this section, we relax the $\mathrm{Max}(P_1, P_2) >> \mathrm{Min}(P_1, P_2)$ condition as required in the previous section. As before, we define m to be $\mathrm{Max}(P_1, P_2)/\mathrm{Min}(P_1, P_2)$ and denote P as the total number of processors and assume $P = \mathrm{Max}(P_1, P_2)$, i.e., the number of processors in the system equals the number of processors required by the high-concurrency stage. For cases when $P > \mathrm{Max}(P_1, P_2)$, a good approximation will be given after the introduction of the Scale-up rule, which will be described in the following section. In this section, we will only consider the discipline which gives the priority to stage 2 with preemption.

A. The LH Job Model

In this subsection, we consider the situation when the low-concurrency stage precedes the high-concurrency stage. This job can be modeled as shown in Fig. 1(a). The difference between this model and the one given in the previous section is that in this model there can be at most $\lceil m \rceil$ jobs working concurrently on the low-concurrency stage if there is no job working on the high-concurrency stage. As before, if there is a job working on the high-concurrency stage, all jobs in the low-concurrency stage will be forced to wait in the queue.

To draw a Markov chain for such a model, since $P_2 = mP_1$, we can normalize the number of processors required by stages by using $P_1 = 1$ and $P_2 = P = m$ without affecting the result. For example, a job model with $P_1 = 2$, $P_2 = 6$, and $P = 6$ should have the same performance with the job model with $P_1 = 1$, $P_2 = 3$, and $P = 3$. We define $p(k, j)$ to be the probability that there are k jobs in stage 1 and j jobs in stage 2 in the system. Since the priority is given to stage 2, hence whenever $j > 0$, then all k jobs in stage 1 have to wait in the queue. Furthermore, all jobs in stage 1 will be paused whenever $j = 1$; hence no more job can advance from stage 1 into stage 2. Hence, the value of j can only be 0 or 1. If $j = 0$ and $k \leq m$, then all jobs in stage 1 are in service. If $j = 0$ and $k > m$, then there are m jobs working on stage 1 and the other $k - m$ jobs are waiting in the queue. Hence, the Markov chain of this model can be shown in Fig. 5.

Note that the overall system load can be expressed as $\lambda(m\mu_1 + \mu_2)/m\mu_1\mu_2$, hence the condition for a stable system is $\lambda < m\mu_1\mu_2/(m\mu_1 + \mu_2)$. We obtain the z-transform of the number of jobs in the system in the following equations.

For the LH job model as shown in Fig. 1(a), the z-transform of the number of jobs in the system is

$$P(z) = \frac{\mu_1\mu_2}{-\lambda^2 z + m\mu_1\mu_2 - m\lambda\mu_1 z - \lambda\mu_2 z + \lambda^2 z^2} \sum_{k=0}^{m-1} (m-k)p(k,0)z^k \tag{9}$$

where $p(k, 0)$ for $0 \leq k \leq m-1$ can be obtained from (10) to (13).

$$\sum_{k=0}^{m} \cdot (m-k)p(k,0) = \frac{m\mu_1\mu_2 - m\lambda\mu_1 - \lambda\mu_2}{\mu_1\mu_2} \tag{10}$$

$$p(1,0) = \frac{\lambda(\lambda+\mu_2)}{\mu_1\mu_2} p(0,0) \tag{11}$$

$$p(2,0) = \frac{\lambda^2[(\lambda+\mu_2)^2 + \lambda\mu_1]}{2\mu_1^2\mu_2^2} p(0,0). \tag{12}$$

For $k \geq 3$

$$p(k,0) = \frac{(\lambda+\mu_2)[\lambda+(k-1)\mu_1]}{k\mu_1\mu_2} p(k-1,0) - \frac{\lambda[\lambda+2(k-1)\mu_1]}{k\mu_1\mu_2} p(k-2,0) - \frac{\lambda^2}{k\mu_1\mu_2} p(k-3,0). \tag{13}$$

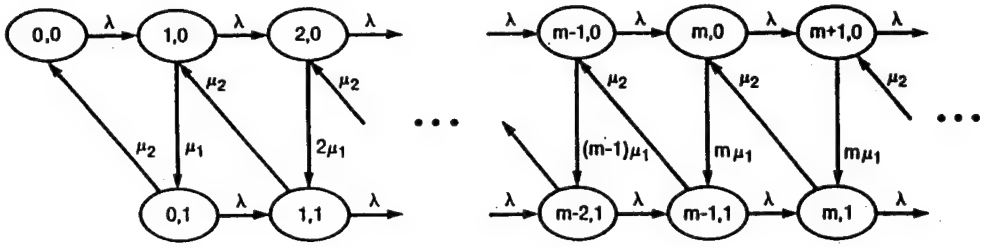


Fig. 5. Markov chain for LH job model.

The proofs of (9) to (13) are included in the Appendix C. From the above results, we can easily find the mean number of jobs in the system. Using the mean number of jobs in the system and Little's result [12], we can find the mean system time as follows.

$$T_{LH} = \frac{\mu_1\mu_2}{\lambda(m\mu_1\mu_2 - m\lambda\mu_1 - \lambda\mu_2)} \sum_{k=1}^{m-1} \cdot k(m-k)p(k,0) - \frac{\lambda - m\mu_1 - \mu_2}{m\mu_1\mu_2 - m\lambda\mu_1 - \lambda\mu_2} \tag{14}$$

where $p(k,0)$ for $0 \le k \le m-1$ can be obtained from (10) to (13).

Here we will compare the results shown in (5) and (14). For both cases, the job model considered are the LH job model. As mentioned in Section III, the results obtained from (5) should be a good approximation to that from (14) when $m >> 1$. Fig. 6 gives the delay performance of an example with $(P_1, P_2, \mu_1, \mu_2) = (1, 20, 1, 1)$ using (14) and another curve derived from (5) using $\mu_1 = \mu_2 = 1$. Fig. 6 shows that these two curves are very close to each other. Intuitively, we know that the results derived from (5) and (14) will be even closer for a larger m. This confirms our assumption in Section III.

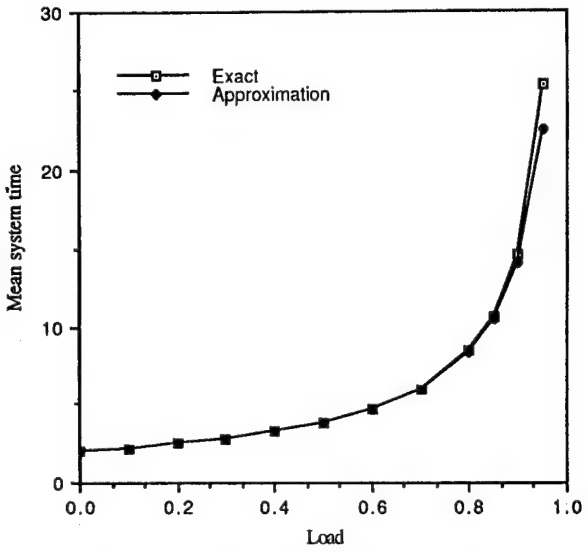


Fig. 6. Comparing (5) and (14) for $m = 20$.

B. The HL Job Model

We study the case when the high-concurrency stage precedes the low-concurrency stage in this subsection. As defined above, we define $p(k,j)$ to be the probability that there are k jobs in stage 1 and j jobs in stage 2 in the system. Since the priority is given to stage 2, hence whenever there are j jobs working in stage 2 (i.e., $j = m = P$), then all k jobs in stage 1 are forced to wait in the queue. Therefore, there can be at most m jobs in stage 2. Unfortunately, we are not able to solve the general case for an arbitrary value of m. However, the result for $m = 2$ can be derived and are given at the bottom of this page. The Markov chain for $m = 2$ is shown in Fig. 7.

To prove (15) is easy but tedious. We omit the details which can be found in [7].

V. Scale-up Rule

The results derived in previous sections are for cases when the number of processors in the system equals the maximum number of processors required by the job. This assumption simplified the Markov chain dramatically, hence making the analysis feasible. However, this analysis is infeasible when the number of available processors is greater than the maximum number of processors required by the job (i.e., $P > \text{Max}(P_1, P_2)$). In order to solve this problem, we propose the *Scale-up Rule* which gives a very good approximation result without adding any analytical complexity. An application of the Scale-up rule to the 2-stage job model when $P > \text{Max}(P_1, P_2)$ is provided in this section. For the rest of the paper, all simulation results are represented by 90% confidence interval using t-distribution and the approach used here is the "replicate/deletion" approach as indicated in [20, p. 551].

There has been considerable effort paid to the analysis of the mean waiting time of an M/G/n queue. Some of them provided bounds [2], [8] while some of them provided approximations [1], [4], [5], [6], [15], [16]. In this paper, we focus our attention of the results obtained in [13] and [14]

$$T_{HL} = \frac{(3\mu_2^2 + \mu_1\mu_2 + 2\mu_1^2)\lambda^2 - \mu_2(\mu_1^2 - \mu_1\mu_2 - 4\mu_2^2)\lambda - 4\mu_2^2(\mu_1^2 + 3\mu_1\mu_2 + 2\mu_2^2)}{\mu_2(\lambda + 2\mu_2)(\mu_1 + 2\mu_2)(\lambda\mu_1 + 2\lambda\mu_2 - 2\mu_1\mu_2)} \tag{15}$$

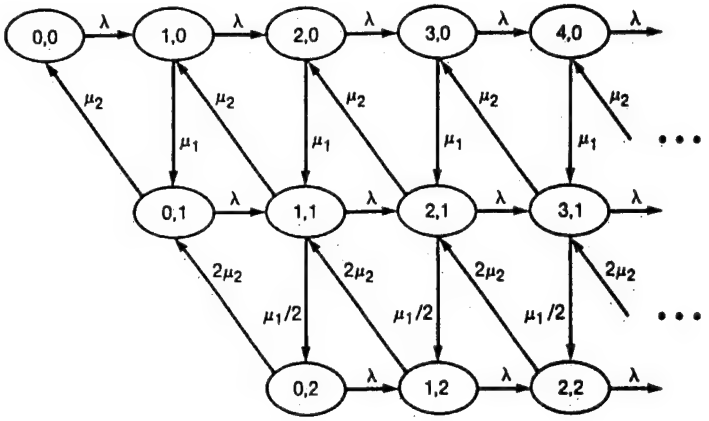


Fig. 7. Markov chain for HL job model.

which provided an approximation for such a system. In [1] and [3] this model was extended to achieve a better result. We define $W_{M/G/n}$ as the mean waiting time for an $M/G/n$ system. In [13] and [14] the following approximation for $W_{M/G/n}$ was suggested as shown in (16). This expression proved to be a very good approximation as some simulation results in [7] demonstrated.

$$W_{M/G/n} \cong \frac{W_{M/G/1}}{W_{M/M/1}} \cdot W_{M/M/n}. \tag{16}$$

A queueing model with a varying number of required processors has been studied in some previous works [17]–[19]. However, in these works, none of them tried to make a full use of all processors. That is, only our model will not allow cases where there are available processors idle in the system while there are also jobs waiting in the queue.

In this subsection, we will extend (16) to the 2-stage job model when $P > \text{Max}(P_1, P_2)$. Although the number of processors required by a job varies during its execution on different stages, we were surprised, and pleased, to discover that the rule which applies to the classical queueing system as stated in (16) also applies in our model. In other words, if we have to find the mean waiting time for a parallel system with $P > \text{Max}(P_1, P_2)$, we will first find the mean waiting time for the system with P' processors where $P' = \text{Max}(P_1, P_2)$ using the method derived in the previous sections. From the result obtained for $P' = \text{Max}(P_1, P_2)$, we apply the following Scale-up rule to obtain the result for system with P processors $(P > P')$.

Scale-up Rule: Given a system with P processors and the job model (P_1, P_2, μ_1, μ_2), we want to find the mean waiting time if $P > \text{Max}(P_1, P_2)$. We first obtain the mean waiting time of the system assuming the number of processors in the system, denoted as P', equals $\text{Max}(P_1, P_2)$. This result can be obtained from the previous section. We denote the mean waiting time for such a system with P' processors as $W_{M/JM/1}(\rho)$, where ρ is the system load and JM stands for "Job Model." For the original system with P processors and denoting P/P' as n $(n > 1)$, we define $W_{M/JM/n}(\rho)$ to be the mean waiting time of the system with $P = nP'$ processors. Similarly, we define $W_{M/M/n}(\rho)$ to be the mean waiting time of an ordinary $M/M/n$ queueing system with mean service time equals $1/\mu_1 + 1/\mu_2$. The Scale-up Rule says that

$$W_{M/JM/n}(\rho) \cong \frac{W_{M/JM/1}(\rho)}{W_{M/M/1}(\rho)} \cdot W_{M/M/n}(\rho). \tag{17}$$

Since the parameters in the right hand side of (17) are all known values, hence $W_{M/JM/n}(\rho)$ in the left hand side can be calculated. The approximation result is a combination of the use of the exact result from Section IV and the use of the Scale-up rule. From the obtained mean waiting time, we can easily obtain the mean system time by adding into the mean service time which equals $1/\mu_1 + 1/\mu_2$. Some examples are given to show how good these approximation results are. Figs. 8 to 11 give a comparison between simulation results and approximation results using the Scale-up rule where the priority is given to stage 2. The approximation results are obtained by first exactly finding the mean system time for $P = 2$ using the technique given in Section IV, then the scale-up rule is used to obtain the results for $P > 2$. Two cases are given for two different job models. One is described as $(P_1, P_2, \mu_1, \mu_2) = (1, 2, 1, 1)$ and the other is described as $(P_1, P_2, \mu_1, \mu_2) = (2, 1, 1, 1)$. Figs. 8 and 9 show the results when $P = 4$ and Figs. 10 and 11 show the result when $P = 10$. From these figures we show that the Scale-up Rule is indeed a very good approximation method.

VI. An Approximation for the General Cases

As we mentioned earlier, to exactly evaluate the performance of a general case with N $(N > 2)$ stages is extremely difficult. In this section, using the exact solution of the 2-stage model and the scale-up rule, we give an approximation method for any general processor-time task graph and any number of processors in the system. In this model, we give higher priority to jobs closer to completion. That is, jobs in the last stage (the Nth stage) have the highest priority while jobs in the first stage have the lowest priority. Again, preemption is assumed. The simulation of this approximation method shows reasonably good results.

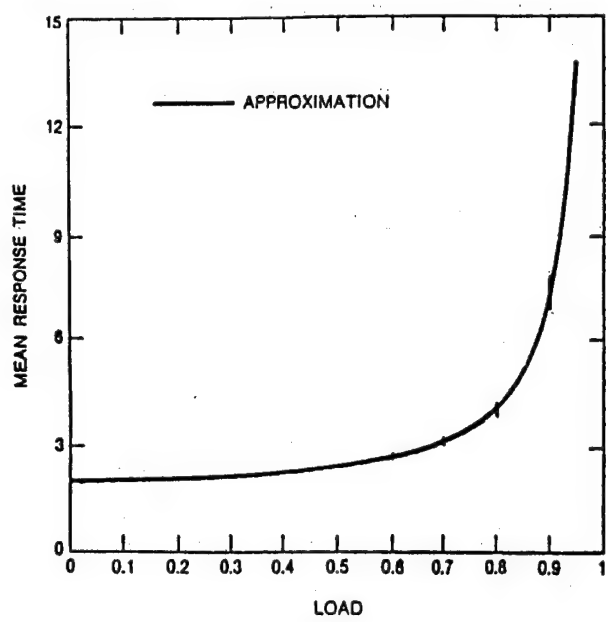


Fig. 8. An approximation result for job model $(1, 2, 1, 1)$ and $P = 4$.

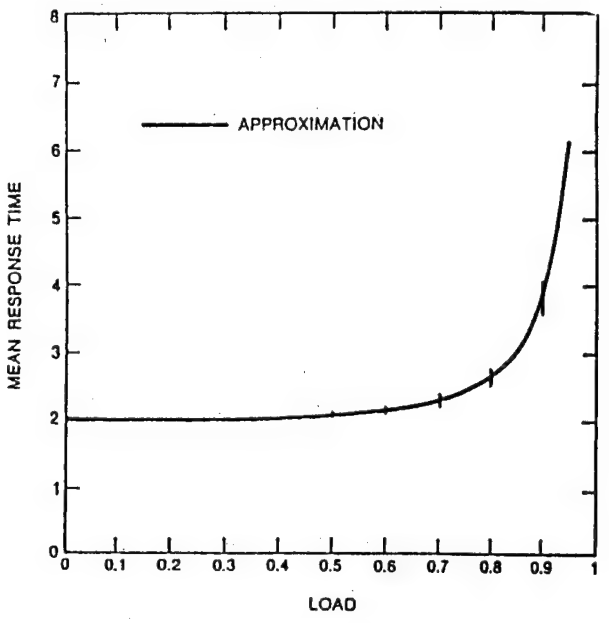


Fig. 10. An approximation result for job model $(1, 2, 1, 1)$ and $P = 10$.

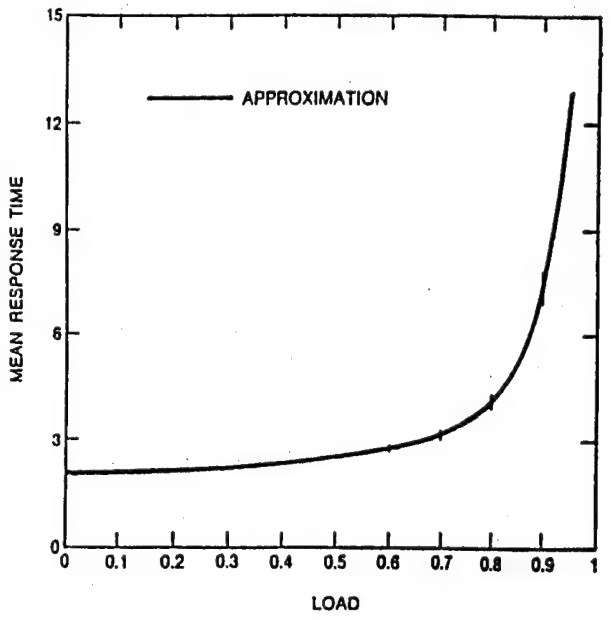


Fig. 9. An approximation result for job model $(2, 1, 1, 1)$ and $P = 4$.

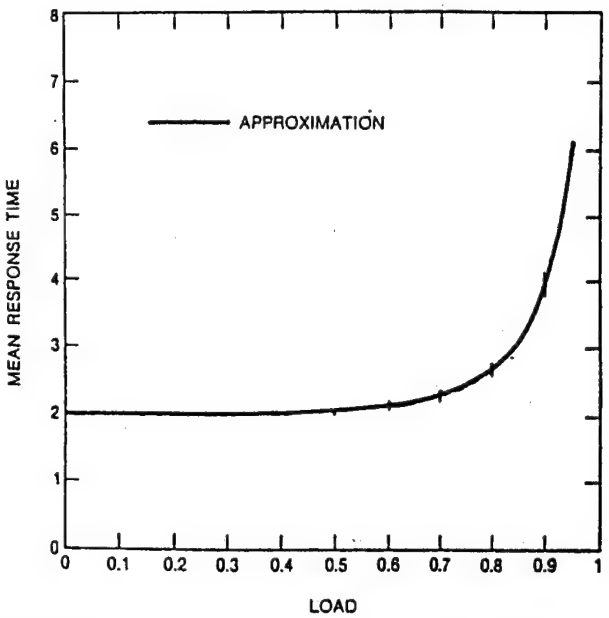


Fig. 11. An approximation result for job model $(2, 1, 1, 1)$ and $P = 10$.

Given a processor-time task graph, we divide the processor-time task graph into two pieces such that the mean service time for each piece is the same. In each piece, we average the work load over its service time to find the *average* number of processors needed. By so doing, we achieve a 2-stage model as analyzed in Section IV. We use this modified 2-stage model as the basis for the approximation model.

If the first stage requires fewer processors than the second stage in the modified model, we use the result obtained in Section IV-A. An example is shown in Fig. 12. In Fig. 12(a), the processor vector for this 4-stage job is $[3, 1, 3, 9]$ and the corresponding time vector is $[1/2, 1/2, 1/2, 1/2]$. By dividing the time vector into two equal pieces, the first two stages of Fig. 12(a) will be merged into the first stage of the modified model as shown in Fig. 12(b). Similarly, the last two stages of Fig. 12(a) will be merged into the second stage of the modified model as shown in Fig. 12(b). In order to make the workload in Fig. 12(b) to be the same as in Fig. 12(a), the processor vector of Fig. 12(b) is $[2, 6]$ and the time vector is $[1, 1]$. Figs. 13 and 14 show the approximation results for this example given 12 and 45 processors in the system, respectively. From these figures, we see that the approximation results are very close to the simulation results.

If the first stage requires more processors than the second stage in the modified model, there is one more step to be done in the approximation method. An example is given in Fig. 15. In Fig. 15(a), the processor vector is $[3, 9, 3, 1]$ and the time vector is $[1/2, 1/2, 1/2, 1/2]$. Applying the same rule as we did in Fig. 12, we convert Fig. 15(a) into Fig. 15(b) such that the processor vector and the time vector of Fig. 15(b)

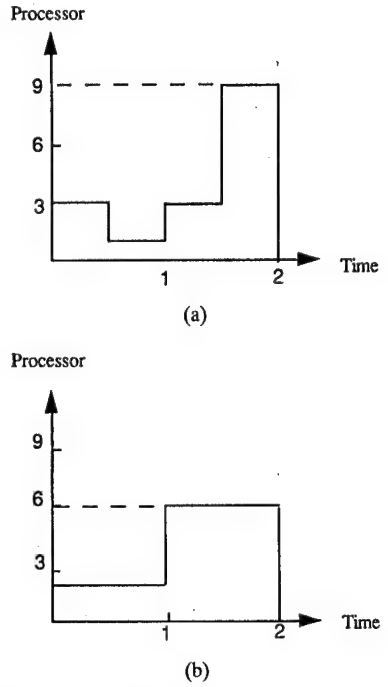


Fig. 12. (a) An example with $[P_1, P_2, P_3, P_4, \mu_1, \mu_2, \mu_3, \mu_4] = [3, 1, 3, 9, 1/2, 1/2, 1/2, 1/2]$. (b) An approximation model for (a).

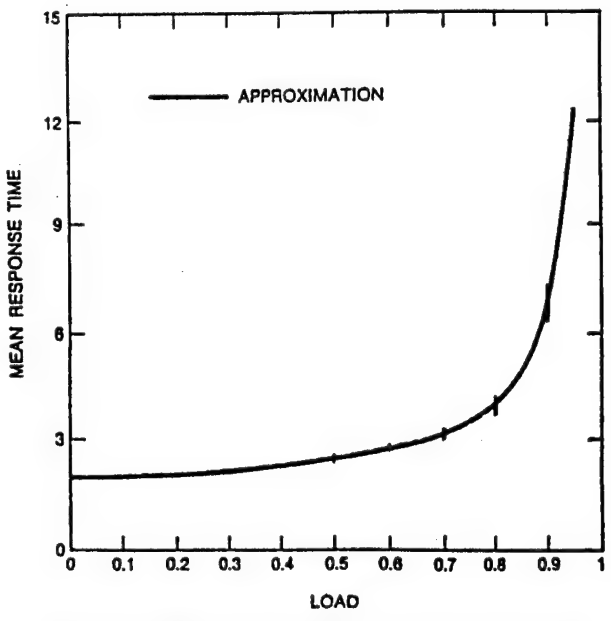


Fig. 13. An approximation result for Fig. 12(a) and $P = 12$.

are $[6, 2]$ and $[1, 1]$, respectively. Since we can analyze the system only when the number of processors required in the first stage is exactly twice of that in the second stage, we have to modify the modified 2-stage model by using a modified 3-stage model. The first stage of the modified 3-stage model [as shown in Fig. 15(c)] is the same as the first stage of the modified 2-stage model [as shown in Fig. 15(b)]. The second stage of the modified 3-stage model is modified such that it requires exactly half of the processors required in the first stage (of the modified 3-stage model) and the total workload required in the stage is the same as that of the second stage

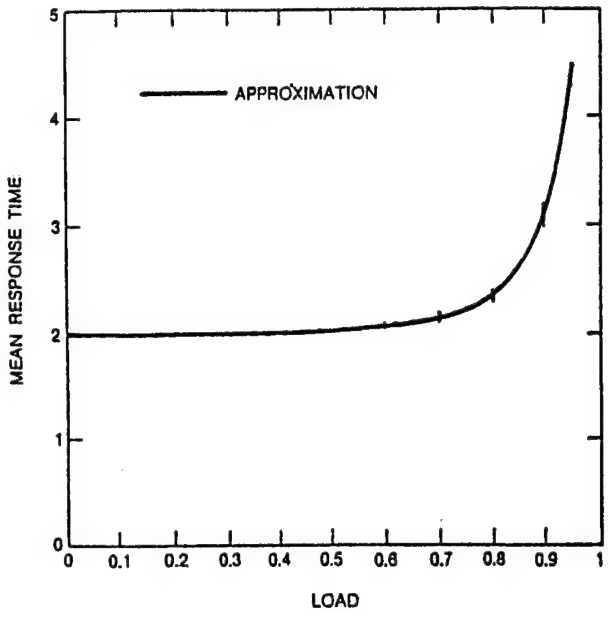


Fig. 14. An approximation result for Fig. 12(a) and $P = 45$.

from the 2-stage model. The third stage in the 3-stage model is used to adjust the no-queueing service time such that it is the same for the modified 2-stage model and the modified 3-stage model.

The processor vector and the time vector for Fig. 15(c) are $[6, 3, 0]$ and $[1, 2/3, 1/3]$, respectively. Although the model in Fig. 15(c) is different from the model described in Section IV-B, it can be solved by using the result from Section IV-B. Notice that the third stage has the highest priority and requires no processors from the system at all; the existence of stage three has no impact on stages one and two. Hence, we can solve this problem by first neglecting the third stage in Fig. 15(c) and apply the results obtained from Section IV-B; from this result, we add the mean service time of stage three to it to get the overall mean response time. Figs. 16 and 17 show the approximation results for this example given there are 12 and 45 processors in the system, respectively.

VII. Conclusions

In this paper we are able to find the average system delay of various job models and disciplines when $\text{Max}(P_1, P_2) \geq P >> 1$ and $\text{Max}(P_1, P_2) >> \text{Min}(P_1, P_2)$ in a large parallel processing system. Further, we achieve the following conclusion that the priority should be given to jobs working on the low-concurrency stage to achieve a better delay performance. This priority assignment scheme remains true for 3-stage job models with a high-concurrency stage preceded and followed by low-concurrency stages. By dropping the $\text{Max}(P_1, P_2) >> \text{Min}(P_1, P_2)$ condition, we also obtain the mean system delays for cases when the priority is given to jobs working on stage 2. A Scale-up Rule is further proposed which gives very good approximation results for systems when the number of processors in the system is greater than $\text{Max}(P_1, P_2)$. Finally, an approximation model for the general cases with N $(N > 2)$ stages is included which shows reasonably good results.

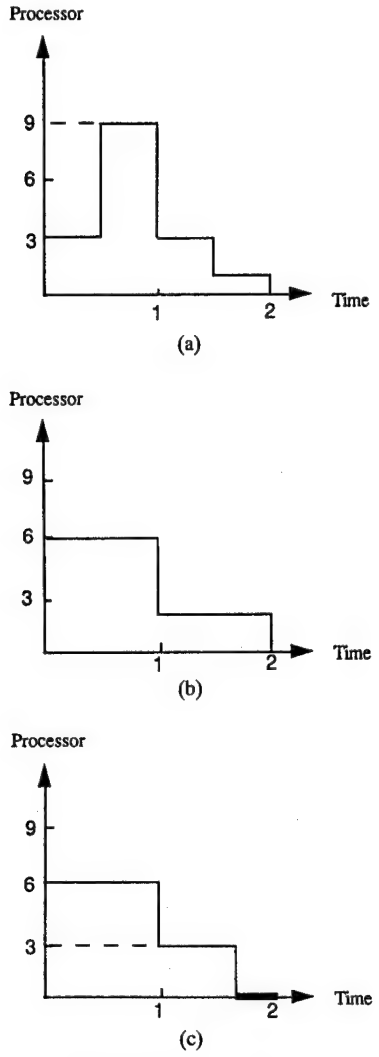


Fig. 15. (a) An example with $[P_1, P_2, P_3, P_4, \mu_1, \mu_2, \mu_3, \mu_4] = [3, 9, 3, 1, 1/2, 1/2, 1/2, 1/2]$. (b) The first step toward approximation model for (a). (c) The second step toward approximation model for Fig. 15(a).

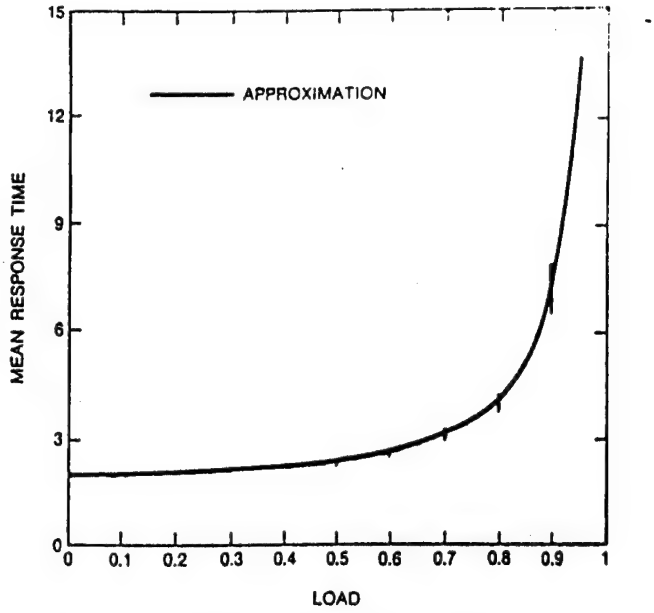


Fig. 16. An approximation result for Fig. 15(a) and $P = 12$.

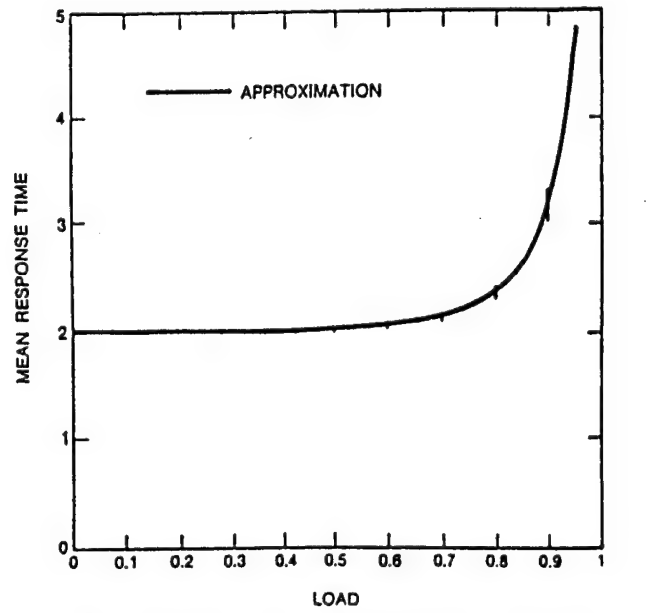


Fig. 17. An approximation result for Fig. 15(a) and $P = 45$.

APPENDIX A

A. Proof of Theorem 1

Proof: For the HL job model, we can regard stage 1 as an M/M/1 queue. Hence, we have the Laplace transform of the system time for stage 1, denoted as $Y_1^*(s)$, given in [9, p. 195]

$$Y_1^*(s) = \frac{\mu_1(1-\rho_1)}{s+\mu_1(1-\rho_1)}. \tag{A1}$$

For stage 2, the system time can be found by using the delay cycle analysis given in [10, p. 110]. When stage 1 is idle, jobs in stage 2 have an exponential service time distribution with parameter μ_2. However, when stage 1 becomes busy, jobs in stage 2 will all be paused until stage 1 becomes idle again and resume the work. Clearly, the system time for stage 2 can be modeled as a delay cycle. The distribution of the initial delay cycle can be found in the following.

It is shown in [9] in page 176 that in an M/G/1 system, the probability that a departure finds k jobs in the system equals the probability that there are k jobs in the system in equilibrium. Hence,

$$Pr[\text{a departure from stage 1 finds } k \text{ jobs still in stage 1}] = (1-\rho_1)\rho_1^k. \tag{A2}$$

A departure from stage 1 to begin receiving service in stage 2 finds k jobs still in stage 1 has an initial delay cycle whose Laplace transformed distribution, denoted as $G_0^{(k)}(s)$, can be represented as

$$G_0^{(k)}(s) = \left(\frac{\mu_1}{s+\mu_1}\right)^k \cdot \frac{\mu_2}{s+\mu_2}. \tag{A3}$$

From the delay cycle analysis formula [10], we have the Laplace transform of the system time for stage 2, denoted as

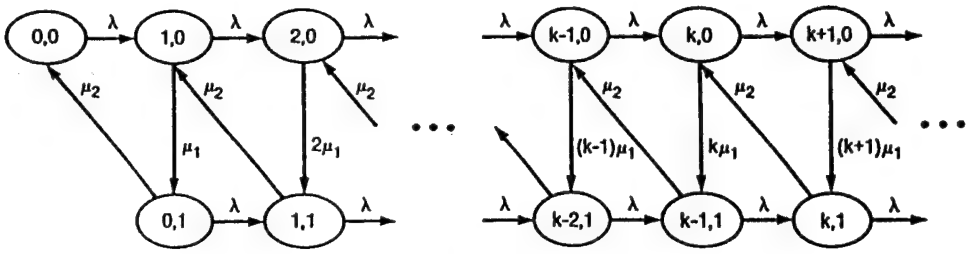


Fig. 18. Markov chain for LH job model.

$Y_2^*(s)$, given as

$$\begin{aligned} Y_2^*(s) &= \sum_{k=0}^{\infty} \cdot (1-\rho_1)\rho_1^k \cdot G_0^{(k)}(s+\lambda-\lambda G^*(s)) \\ &= \sum_{k=0}^{\infty} \cdot (1-\rho_1)\rho_1^k \cdot \left[\frac{\mu_1}{s+\lambda-\lambda G^*(s)+\mu_1}\right]^k \\ &\quad \cdot \frac{\mu_2}{s+\lambda-\lambda G^*(s)+\mu_2} \\ &= \mu_2(1-\rho_1) \cdot \frac{1}{s+\lambda-\lambda G^*(s)+\mu_2} \\ &\quad \cdot \frac{s+\lambda-\lambda G^*(s)+\mu_1}{s-\lambda G^*(s)+\mu_1} \\ &= \frac{\mu_1\mu_2(1-\rho_1)}{G^*(s)[s+\lambda-\lambda G^*(s)+\mu_2][s-\lambda G^*(s)+\mu_1]} \end{aligned}$$

where ρ_1 equals λ/μ_1 and $G^*(s)$ is the Laplace transform of the busy period distribution of stage 1.

From (A1), (A2), and (A3), the Laplace Transform, $Y_{\mathrm{HL}}^*(s)$, of the system time of the HL job model equals $Y_1^*(s) \cdot Y_2^*(s)$ as shown in (1). Q.E.D.

B. Proof of (4)

Proof: By defining state (n_1, n_2) as n_1 jobs in stage 1 and n_2 jobs in stage 2, we have the Markov chain given in Fig. 18 and the following equilibrium balance equations:

$$(\lambda + k\mu_1)p(k,0) = \lambda p(k-1,0) + \mu_2 p(k,1) \qquad k \geq 1 \tag{B1}$$

$$(\lambda + \mu_2)p(k-1,1) = k\mu_1 p(k,0) + \lambda p(k-2,1) \qquad k \geq 2 \tag{B2}$$

$$\lambda p(0,0) = \mu_2 p(0,1) \tag{B3}$$

$$(\lambda + \mu_2)p(0,1) = \mu_1 p(1,0). \tag{B4}$$

We define $p(k)$ to be the probability that there are totally k jobs in the system and define the following notation:

$$P(z) = \sum_{k=0}^{\infty} p(k)z^k, \qquad P_0(z) = \sum_{k=0}^{\infty} p(k,0)z^k,$$

$$P_1(z) = \sum_{k=0}^{\infty} p(k,1)z^k.$$

Since $p(k) = p(k,0) + p(k-1,1)$, it can easily be shown that $P(z) = P_0(z) + zP_1(z)$. From (B1) and (B3) we have

$$\lambda P_0(z) + \mu_1 z \frac{d}{dz} P_0(z) = \lambda z P_0(z) + \mu_2 P_1(z). \tag{B5}$$

From (B2) and (B4) we have

$$(\lambda + \mu_2)P_1(z) = \mu_1 \frac{d}{dz} P_0(z) + \lambda z P_1(z). \tag{B6}$$

From (B5) and (B6) we have the following differential equation for $P_0(z)$.

$$(\lambda z - \mu_2)\mu_1 \frac{d}{dz} P_0(z) + \lambda(\lambda + \mu_2 - \lambda z)P_0(z) = 0.$$

Solving this linear differential equation we obtain the following explicit expression of $P_0(z)$:

$$P_0(z) = c \cdot \exp((\lambda/\mu_1)(z - \ln|\lambda z - \mu_2|)) \tag{B7}$$

where c is a constant yet to be determined. From (B5) and (B6) we have

$$P_1(z) = \frac{\lambda}{\mu_2 - \lambda z} P_0(z). \tag{B8}$$

Combining (B7), (B8), and $P(1) = 1$ we find c to be

$$c = \frac{\mu_2 - \lambda}{\mu_2} \exp((-\lambda/\mu_1)(1 - \ln(\mu_2 - \lambda))).$$

From the above results we have the z-transform of the number of jobs in the system as shown in (4). Q.E.D.

C. Proof of (9) to (13)

Proof: From the Markov chain given in Fig. 6, we have

$$(\lambda + k\mu_1)p(k,0) = \lambda p(k-1,0) + \mu_2 p(k,1) \qquad 1 \leq k \leq m \tag{C1}$$

$$(\lambda + m\mu_1)p(k,0) = \lambda p(k-1,0) + \mu_2 p(k,1) \qquad k > m \tag{C2}$$

$$(\lambda + \mu_2)p(k-1,1) = \lambda p(k-2,1) + k\mu_1 p(k,0) \qquad 2 \leq k \leq m \tag{C3}$$

$$(\lambda + \mu_2)p(k-1,1) = \lambda p(k-2,1) + m\mu_1 p(k,0) \qquad k > m \tag{C4}$$

$$\lambda p(0,0) = \mu_2 p(0,1) \tag{C5}$$

$$(\lambda + \mu_2)p(0,1) = \mu_1 p(1,0). \tag{C6}$$

We define $p(k) = \Pr[k \text{ jobs in the system}] = p(k,0) + p(k-1,1)$. We further define

$$P(z) = \sum_{k=0}^{\infty} p(k) z^k, \qquad P_0(z) = \sum_{k=0}^{\infty} p(k,0) z^k,$$

$$P_1(z) - \sum_{k=0}^{\infty} p(k,1) z^k.$$

Hence, we have $P(z) = P_0(z) + zP_1(z)$. From (C1) and (C2) we have

$$(\lambda + m\mu_1 - \lambda z) P_0(z) = \mu_2 P_1(z) + \mu_1 \sum_{k=0}^{m-1} \cdot (m-k) p(k,0) z^k. \quad \text{(C7)}$$

Similarly, from (C3) and (C4) we have

$$[(\lambda + \mu_2) z - \lambda z^2] P_1(z) = m\mu_1 P_0(z) - \mu_1 \sum_{k=0}^{m-1} \cdot (m-k) p(k,0) z^k. \quad \text{(C8)}$$

From (C7) and (C8) we have

$$P_1(z) = \frac{\lambda}{\mu_2 - \lambda z} P_0(z). \quad \text{(C9)}$$

From (C7), (C8), and (C9) we have $P(z)$ as shown in (9). The remaining job is to find all $p(k,0)$ for $0 \le k \le m-1$. Using (9) and $P(1) = P_0(1) + P_1(1) = 1$, we can obtain (10). Equations (11), (12), and (13) can easily be obtained from arranging (C1), (C3), (C5), and (C6). From (10) through (13) we are able to find all $p(k,0)$ for $0 \le k \le m-1$. Q.E.D.

REFERENCES

[1] O. J. Boxma, J. W. Cohen, and N. Huffels, "Approximations of the mean waiting time in an M/G/c queueing system," *Oper. Res.*, vol. 27, pp. 1115–1127, 1979.

[2] S. L. Brumelle, "Some inequalities for parallel server queues," *Oper. Res.*, vol. 19, pp. 402–413, 1971.

[3] G. P. Cosmetatos, "Some approximation equilibrium results for the multi-server queue (M/G/r)," *Opnl. Res. Quart.*, vol. 27, pp. 615–620, 1976.

[4] C. D. Crommelin, "Delay probability formulae," *P.O. Elec. Eng. J.*, vol. 26, pp. 266–274, 1934.

[5] F. S. Hillier and F. D. Lo, "Tables for multiple-server queueing systems involving Erlang distribution," Tech. Rep. 31, Dep. of Oper. Res., Stanford Univ., 1971.

[6] M. H. van Hoorn and H. C. Tijms, "Approximations for the waiting time distribution of the M/G/c queue," *Perform. Eval*, vol. 2, pp. 22–28, 1982.

[7] J. Huang, "On the behavior of algorithms in a multiprocessing environment," Ph.D. dissertation, Comput. Sci. Dep. U.C.L.A., 1988.

[8] J. F. C. Kingman, "Inequalities in the theory of queues," *J. Royal Statistical Society, Series B*, vol. 32, pp. 102–110, 1970.

[9] L. Kleinrock, *Queueing Systems, Vol. 1: Theory.* New York: Wiley-Interscience, 1975.

[10] ——, *Queueing Systems, Vol. 2: Computer Applications.* New York: Wiley-Interscience, 1976.

[11] ——, "Performance models for distributed systems," in *Teletraffic Analysis and Computer Performance Evaluation*, Proc. International Seminar held at the centre for Mathematics and Computer Science (CWI), Amsterdam, The Netherlands, June 2–6, 1986, pp. 1–15.

[12] J. D. C. Little, "A proof of the queueing formula $L = \lambda W$," *Oper. Res.*, vol. 9, pp. 383–387, 1961.

[13] E. Maaloe, "Approximation formulae for estimation of waiting-time in multiple-channel queueing system," *Mgmt. Sci.*, vol. 19, pp. 703–710, 1973.

[14] S. A. Nozaki and S. M. Ross, "Approximations in finite-capacity multi-server queues with poisson arrivals," *J. Appl. Probability*, vol. 15, no. 4, pp. 826–834, Dec. 1978.

[15] R. Syski, *Introduction Congestion Theory in Telephone Systems*, second ed. Amsterdam, The Netherlands: North Holland, 1984.

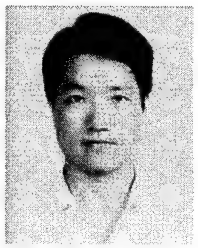
[16] Y. Yakahashi, "An approximation formula for the mean waiting time of an M/G/m queue," *J. Oper. Res. Soc. Japan*, vol. 20, pp. 150–163, 1977.

[17] Y. De Serres and L. G. Mason, "A multiserver queue with narrow- and wide-band customers and wide-band restricted access," *IEEE Trans. Commun.*, vol. 36, no. 6, pp. 675–684, June 1988.

[18] B. Kraimeche and M. Schwartz, "Bandwidth allocation strategies in wide-band integrated networks," *IEEE J. Select. Areas Commun.*, vol. SAC-4, pp. 869–878, Sept. 1986.

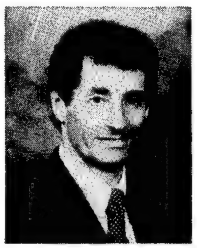
[19] L. Green, "A queueing system in which customers require a random number of servers," *Oper. Res.*, vol. 28, no. 6, pp. 1335–1346, Nov.–Dec. 1980.

[20] A. M. Law and W. D. Kelton, *Simulation Modeling & Analysis*, second ed. New York: McGraw-Hill, 1991.



Jau-Hsiung Huang (A'89) received the B.S. degree in electrical engineering from National Taiwan University in 1981, and the M.S. and Ph.D. degrees in computer science from the University of California, Los Angeles, in 1985 and 1988, respectively.

He is Associate Professor of Department of Computer Science and Information Engineering at National Taiwan University, Taipei, Taiwan. He joined the faculty at National Taiwan University in 1988. His research interests include design and performance evaluation of high speed networks, multimedia systems, and parallel and distributed systems.



Leonard Kleinrock (S'55–M'64–SM'71–F'73) received the B.S. degree in electrical engineering from the City College of New York in 1957 (evening session) and the M.S.E.E. and Ph.D.E.E. degrees from the Massachusetts Institute of Technology in 1959 and 1963, respectively.

He is Professor and Chairman of Computer Science at the University of California, Los Angeles. While at M.I.T., he worked at the Research Laboratory for Electronics, as well as with the computer research group of Lincoln Laboratory in advanced technology. He joined the faculty at U.C.L.A. in 1963. His research interests focus on performance evaluation of high speed networks and parallel and distributed systems. He has had over 170 papers published and is the author of five books–*Communication Nets: Stochastic Message Flow and Delay*, 1964; *Queueing Systems, Volume I: Theory*, 1975; *Queueing Systems, Volume II: Computer Applications*, 1976; *Solutions Manual for Queueing Systems, Volume I*, 1982, and most recently, *Solutions Manual for Queueing Systems, Volume II*, 1986. He is a well-known lecturer in the computer industry. He is the principal investigator for the DARPA Parallel Systems Laboratory contract at U.C.L.A.

Dr. Kleinrock is a member of the National Academy of Engineering, is a Guggenheim Fellow, and a member of the Computer Science and Technology Board of the National Research Council. He has received numerous best paper and teaching awards, including the ICC 1978 Prize Winning Paper Award, the 1976 Lanchester Prize for outstanding work in Operations Research, and the Communications Society 1975 Leonard G. Abraham Prize Paper Award. In 1982, as well as having been selected to receive the C.C.N.Y. Townsend Harris Medal, he was co-winner of the L. M. Ericsson Prize, presented by His Majesty King Carl Gustaf of Sweden, for his outstanding contribution in packet switching technology. In July of 1986, he received the 12th Marconi International Fellowship Award, presented by His Royal Highness Prince Albert, brother of King Baudoin of Belgium, for his pioneering work in the field of computer networks. In the same year, he received the U.C.L.A. Outstanding Teacher Award. In 1990, he received the ACM SIGCOMM award recognizing his seminal role in developing methods for analyzing packet network technology. He was a co-founder of Linkabit Corporation. He is also the founder and CEO of Technology Transfer Institute, a computer/communications seminar, conference and consulting organization located in Santa Monica, CA.

Collecting Unused Processing Capacity: An Analysis of Transient Distributed Systems

Leonard Kleinrock, *Fellow, IEEE,* and Willard Korfhage, *Member, IEEE*

Abstract—**Distributed systems frequently have large numbers of idle computers and workstations. If we could make use of these, then considerable computing power could be harnessed at low cost. We analyze such systems using Brownian motion with drift to model the execution of a program distributed over the idle computers in a network of idle and busy processors, determining how the use of these "transient" processors affects a program's execution time. We find the probability density of a programs finishing time on both single and multiple transient processors, explore these results for qualitative insight, and suggest some approximations for the finishing time probability density that may be useful.**

Index Terms— **Brownian motion, distributed processing, idle processors, performance analysis, transient processors.**

I. Introduction

DISTRIBUTED systems frequently have large numbers of idle computers and workstations. If we could use these, then considerable computing power could be harnessed at low cost. In this paper, we model program execution on a network of workstations, some idle and some not. Because we use only the idle time on the processors, they are not always available for use. Hence, we call these machines "transient" processors. Using a direct analysis and a cumulative alternating renewal process analysis both provide simple expressions for the probability density of the program finishing time on a single transient processor. We extend the cumulative alternating renewal process into a Brownian motion with drift model of the finishing time density on multiple processors. We then examine the properties of the finishing time probability density to achieve some qualitative insight into the results. Finally, we suggest several approximations for the finishing time probability density, and discuss under what conditions they can be used.

A. Background

Networks of computers are common in business and research environments throughout the world. Local area networks, which were originally introduced to ease data and device sharing, have grown in speed, sophistication, and size to the point that effective distributed processing can be performed on them. These networks vary in size from a handful of personal computers on a low-speed network, to networks consisting of thousands of workstations and a variety of larger machines on a high-speed, fiber-optic network. As a typical example, consider a network of workstations on a high-speed network in a research laboratory. Not only are there many machines, well connected by the network, but the users are likely to demand more and more computing power as the applications grow.

Networks of workstations have grown in spite of theoretical considerations that would discourage them. It is well known in queueing theory [2] that a single server of large capacity shared by many users provides better response time than many smaller servers with the same total capacity. Thus we might expect that a mainframe will provide faster service to its users than a network of workstations. Of course, one may argue that for the same amount of money one can buy much more workstation capacity than mainframe capacity, but even then we would like to make the best use possible of our computing resources. This implies that a network of workstations would have improved performance (overall) if the workstations were considered part of one processor pool, available for the execution of all programs. Some systems, such as Amoeba [3] provide a pool of processors strictly as compute servers. We, however, would like to retain the usual use of workstations on the network in addition to considering them part of the processor pool.

The solution to this is idle time. On these networks, we often have the situation that many of the personal computers and workstations are sitting idle, waiting for their users, and thus being wasted ([4], [5]). If we could recover this wasted time for useful processing, then we would have considerable computing power available to us at low cost. We refer to these processors, which are sometimes busy and sometimes not, as *transient* processors.

Whether this is technically feasible or not depends on a variety of factors, such as the properties of the communications medium, the properties of the computers, and the statistical characteristics of the user population.[1] In all systems of this type, one concern is that the "owner" of a machine should not see any degradation in performance because of the background programs. Any background computation should be aborted

Manuscript received July 1, 1990; revised July 29, 1991. This work was supported by the Defense Advanced Research Projects Agency under Contract MDA 903-82-C0064, Advanced Teleprocessing Systems, and Contract MDA 903-87-C0663, Parallel Systems Laboratory. W. Korfhage was also supported by an HP/AEA fellowship. This work was done while W. Korfhage was at U.C.L.A.

L. Kleinrock is with the Department of Computer Science, University of Calfiornia, Los Angeles.

W. Korfhage is with the Department of Computer Science, Polytechnic University, Brooklyn, NY 11201.

IEEE Log Number 9208484.

[1]There are also other important but nontechnological factors, such as people's resistance to the use of "their" machine, that would determine if and how a distributed system would be implemented. This paper does not examine such matters.

when user activity is detected, and not restarted until the system is sure that the machine is idle.

There has been much research on systems that make use of idle workstations through load balancing and process migration; we mention here a few such systems. Most of these provide remote program execution facilities (e.g., run a compiler at another machine rather than on the local workstation), rather than specifically supporting distributed computations. We make note of those systems that have been designed specifically for distributed computations.

Alonso and Cova [6] discuss a load balancing system for workstations in which a workstation tries to execute a job remotely if the local processor load is greater than some "High Mark," and accepts jobs from other workstations if the local processor load is less than some "Low Mark." This allows for a continuum of migration policies.

At U.C.L.A., the Benevolent Bandit Laboratory (BBL) [7] runs distributed computations under MS-DOS on a network of IBM PC-AT's. A special shell runs on each machine, and when a machine is at the operating system prompt level (as opposed to running a program), it is available for use. If someone starts to use a machine currently part of a background, distributed computation, the system can select and start a replacement machine from the pool of idle processors. Because the system was also intended for the investigation of distributed algorithms, special features, such as the ability to mimic any connection topology, and some distributed debugging facilities, have been built in.

Lyle and Lu [8] describe a simple remote program execution facility that operates at the shell level, like BBL, rather than the kernel level. This has the advantage of being simple to implement, yet still uses idle workstations.

Condor ([9], [10], [4]) is a very successful remote program execution facility running on workstations at the University of Wisconsin. The developers of the system have made a number of useful measurements of workstation behavior in [10].

The Butler system [11], running on Andrew workstations at Carnegie-Mellon also provides remote program execution facilities. The system uses this to run *gypsy servers*, which are network servers that run on idle workstations instead of on a fixed machine.

Stumm [12] discusses a remote program execution and task migration facility for the V kernel. His paper discusses various issues, such as the migration policy, and offers thoughts on using the system for distributed computations.

The *Worm* program [13] was developed at Xerox PARC as an experiment in distributed processing. Worms prowled the network, collecting idle workstations and using them to perform some action, typically displaying a message or running a diagnostic program.

There have also been ad-hoc attempts to use the idle time on processors. Dr. Tim Shimeall [14], during his dissertation research, wrote a program "polite" that ran a software analysis program on workstations when no one was logged in and suspended the program when the workstation was being used. He reports that he finished nearly 10 CPU years of work in about 6 months on 20 workstations using this program. But again, this was very much a simple remote job execution facility, put together out of need, and it was never analyzed.

B. Outline

Section II discusses our model of the network. Section III gives a simple analysis of the average time for a program to finish. Sections IV and V then develop three models of the network and analyze them to find the distribution of time to finish a fixed amount of work.

The first model, in Section IV-B, is a single processor model with general *available* and *nonavailable* times. We examine the number of nonavailable periods interrupting a program, and from this we find the Laplace transform of the distribution of the time to finish a program (response time), and then the mean and variance of the response time.

The second model, in Section IV-C, is also a single processor model, but it is analyzed as a cumulative, alternating renewal process. We find that the asymptotic distribution of the accumulated work (over a long period of time) is Gaussian, with simple expressions for the mean and variance.

The third model, in Section V-A, handles multiple processors and views the amount of work done over time as Brownian motion with drift. We scale the asymptotic mean and variance of the accumulated work from the second, single processor model to the case of M processors, and use this as the mean and variance of the Brownian motion with drift. From this we get the probability density of the time to finish a fixed amount of work on M processors. The mean and the variance agree very closely, for $M = 1$, with the first model.

Finally, Section VI contains the conclusions. These three models offer an approach to predicting performance of distributed programs on transient processors. By relaxing some of our assumptions, more sophisticated models could be derived from those described here.

II. The Model of the Network and the Workload

A. The Network

Assume that we have a network of M identical processors, each of which has a capacity to complete one minute of work per minute. A processor alternates between a *nonavailable state* (signified by *n* or *na*), when the owner is using it (e.g., typing at the keyboard), and an *available state* (signified by *a* or *av*), when it is sitting idle. The lengths of nonavailable periods are independent and identically distributed (i.i.d.) random variables from distribution $N(t)$, with mean t_n, variance σ_n^2, and corresponding density $n(t)$; we allow any general distribution for $N(t)$, unless otherwise specified. Likewise, available periods are i.i.d. random variables from a general distribution $A(t)$ with mean t_a, variance σ_a^2, and density $a(t)$. The available and nonavailable periods are mutually independent.

B. The Distributed Program Workload

We model a program as consisting of multiple stages of work, each of which must be completed before the start of the next, and each of which represents a deterministic amount

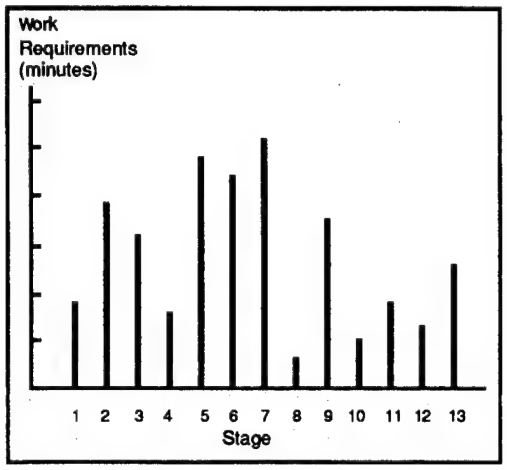


Fig. 1. Execution time profile of an algorithm.

of work (Fig. 1). The time to finish a program is the sum of the times to complete the individual stages. We assume that the time to finish a stage depends only on the amount of work in that stage, and is independent of the other stages. This means that the probability distribution of the *total* time to finish a program is the convolution of the distributions of the individual stage finishing times. Assuming that the network characteristics do not change during the execution of the program, then from the analysis of the time to finish a single stage requiring W minutes of work we can find the finishing time probability density function of all other stages, and from there, the finishing time probability density function of the program as a whole.

We make the simplifying assumption that the work in any stage is infinitely divisible—it can always be divided evenly among all available processors. Note, however, that this assumption does not always obscure program behavior. Some programs, such as the simulators used for the models of this paper, are, in fact, composed of very many independent tasks, and work is always available for any idle processor. In such cases, this assumption captures the program behavior and is not a simplification at all. In addition, in a system with multiple users on many machines, the aggregation of independent jobs, each with a number of tasks, from many users will yield an overall workload that tends to look like many independent tasks (or, at least, enough tasks to keep idle machines busy), and this would fit well with our assumption. We will explore this more in future work.

Another simplifying assumption we make is to ignore overhead that occurs in a real system (e.g., communication delays, processing delays), and thus our model provides an optimistic bound on system performance. Some techniques for removing this assumption are mentioned in [1].

C. What We Seek

The purpose of this paper is to find $f(t)$, the probability density function (pdf) of a program's finishing time (i.e., response time). Also of interest are its mean, $\bar{f}$, and its variance, σ_f^2.

In the process of finding these, we will need another function, namely, the pdf of the amount of work accumulated (i.e., completed) by a processor or network of processors over time. We denote this by $y(u \mid t)$, the probability density that after t minutes of time have elapsed, the processor (or processors) under consideration has accumulated u minutes of work. This function has mean $\mathrm{E}[y(u \mid t)]$ and variance $\mathrm{Var}[y(u \mid t)]$.

D. Notation

We use the abbreviations "PDF" to stand for "probability distribution function." Typically we use capital letters for a PDF and the corresponding lower case letters for a pdf. If $F(x)$ is a PDF, then its pdf is $f(x) = (\partial/\partial x)F(x)$.

E. Example Parameters

Mutka and Livny, [10], made actual measurements of a network of transient processors, and they developed models for the available and nonavailable period densities to fit these measurements. From their results, we derive two examples that we use throughout this paper.

The model they used for the available time PDF was a 3-stage hyperexponential distribution:

$$\begin{aligned} A(t) &= \mathrm{P}\ [\text{length of an available period} \le t] \\ &= 0.33(1 - e^{-(t/3)}) + 0.4(1 - e^{-(t/25)}) \\ &\quad + 0.27(1 - e^{-(t/300)}) \qquad t \ge 0 \end{aligned} \tag{1}$$

which has mean $t_a = 91$ min and variance $\sigma_a^2 = 40225$ min^2.

For the nonavailable time distribution, $N(t) = \mathrm{P}[\text{length of a nonavailable period} \le t]$, they used a shifted 2-stage hyperexponential distribution:

$$N(t) = \begin{cases} 0.7(1 - e^{-(t/7)}) + 0.3(1 - e^{-(t/55)}) & \text{if } t \ge 7 \\ 0 & \text{if } 0 \le t < 7 \end{cases} \tag{2}$$

which has mean $t_n = 31.305$ min and variance $\sigma_n^2 = 2131.83$ min^2. The 7 min shift in the distribution arises because a processor was not declared idle until 7 idle minutes had elapsed.

We use Mutka and Livny's distributions wherever possible in our examples, but frequently we assume exponentially distributed available and nonavailable periods; at such times, we take the means of these exponential distributions to be the numbers given above. The use of exponential distributions instead of hyperexponential distributions will not affect any means that we derive, but any variances that we find will be lower than if we had used Mutka and Livny's distributions.

Regardless of which distributions we use, we take $W = 1000$ min and $M = 1$ for single processor examples, and, $W = 10000$ min (almost 7 days) and $M = 100$ for most multiple processor examples. The reason for the large values of W is explained below.

F. Related Work

One approach to analyzing a single processor system is to use queueing with vacation as a model. In such a system, the

queueing server is subject to randomly occurring stoppages lasting for random amounts of time. There are many varieties of such systems depending upon what restrictions we put on the vacations (see [15] for a survey). For our model, we require vacations to occur preemptively and at any time (as opposed to vacations that occur only when the processor is busy). The earliest analysis of such systems is in [16], and later in [17], but Gaver ([18]) derives the Laplace transform for the finishing time by assuming exponentially distributed available periods and generally distributed nonavailable periods. Federgruen and Green ([19]) extend the analysis to generally distributed available periods, but they find only the first two moments of the finishing time, and not its distribution.

For both single and multiple processor systems, performability analysis offers an alternative approach to ours. Performability analysis ([20], [21], [22]) combines dependability analysis with performance measures. A system is modeled as a Markov or semi-Markov process in which each state of the process represents a possible configuration of the system with respect to failed and working components. In a multiprocessor, for example, the state could be the number of working processors. This state represents the reliability aspect of performability analysis. Associated with each state is a reward representing either the performance measure of interest, or a quantity that may be used to calculate the performance measure. Applied to the models of this paper, the state of the system represents the number of available processors, and the reward for each state is the amount of available computing power (in operations per time unit) in that state. Our goal would be to find the distribution of time it takes for the accumulated reward (accumulated work) to reach a threshold representing the amount of work a program requires.

A number of researchers have examined the problem of finding the distribution of accumulated reward (also known as the performability distribution). Nonrepairable systems, in which a nonavailable processor cannot become available ("be repaired"), are easiest to analyze, but are not applicable to our problem. For repairable systems, some researchers have found methods to get the moments of the performability distribution ([23], [24], [25]), and other researchers have expressed the performability distribution as a double Laplace transform ([24], [26], [23], [27], [28], [29]). In the latter, typically the transform can be inverted analytically on one variable, then inverted numerically on the other variable, although [24] perform the inversion entirely numerically. In [30], de Souza e Silva and Gail apply randomization techniques to numerically find the distribution of performability over a finite time interval.

However, we do not want this distribution of accumulated reward itself, but we would like to find the distribution of time for the accumulated reward to reach a threshold. Because our reward represents accumulated work, this latter distribution is equivalent to the distribution of time to complete a job. In one of the early papers on performability, Beaudry [20] defines this quantity, but never derives it for systems of interest to us. Kulkarni, Nicola, Smith, and Trivedi [27] find a double transform of the job completion time distribution in terms of a system of equations, and provide an algorithm for numerical inversion of the transform. The difference between this previous work and the work contained in this paper is that our model takes a different view of the problem and involves simple, approximate analytical expressions, with no numerical techniques being necessary.

Finally, much of the work on performability concerns itself with transient analysis, because in a well-designed, fault-tolerant system, faults will be quite infrequent, and steady state analysis can be misleading. Our situation is the opposite, with "faults" (processor nonavailability) occurring frequently, and we expect many "faults" to occur before a program finishes execution. Iyer *et al.* [26] note that asymptotically, after a long enough time that every state of the system has been entered many times, the performability distribution is normally distributed, and the mean and standard deviation of this distribution can be found by solving sets of linear equations. In this paper, we come to the same conclusion about the normality of the asymptotic distribution by starting from the analysis of a single processor, as discussed in Sections IV-C and V-A, and in doing so we find simple expressions for the mean and variance of this distribution [(18) and (19)].

One appealing aspect of performability models is that by setting the rewards appropriately for each system state, the model could capture some of the inefficiencies that occur as a program executes on varying numbers of processors, examples of which are the additional communication overhead involved, or the program's inability to use all available processors. These rewards, however, would be specific to that particular program. Ammar and Islam [24] have done this using Generalized Stochastic Petri Nets to generate the reliability model for a specific architecture, and then trace-driven simulations of a specific algorithm to determine the reward for every state of the reliability model. The reward is the inverse of the total execution time of the computation, given the system is in a particular state. Because there may be many states in the reliability model, and hence many simulation runs required, their model is potentially quite time consuming. We are investigating methods by which we can capture the interaction of the algorithm with the architecture within our Brownian motion model.

III. Time to Finish a Program: Quick Means

With simple reasoning, we can find the mean cumulative work over time, $\mathrm{E}[y(u \mid t)]$, and the mean finishing time for a program, $\overline{f}$. Over a long period of time, a processor is available a fraction of the time $p_a = t_a/(t_a + t_n)$, and nonavailable the remaining fraction of the time, $p_n = t_n/(t_a + t_n)$. Over a period of t seconds, the amount of work a processor does is equal to the fraction of time it is available (assuming that there is a large amount of work to do, and that the processor never goes idle), and thus we have the equilibrium approximation

$$\mathrm{E}[y(u \mid t)] = \frac{t_a}{t_a + t_n}\, t. \tag{3}$$

Similarly, it takes $(t_a + t_n)/t_a$ seconds to accumulate one second of work, so the average finishing time for a program

on a single processor is

$$\overline{f} = \frac{t_a + t_n}{t_a} W. \tag{4}$$

In an M processor network, we accumulate work M times faster and thus finish in $(1/M)$th of the time. Therefore:

$$\mathrm{E}[y(u \mid t)] = \frac{t_a M}{(t_a + t_n)} t. \tag{5}$$

$$\overline{f} = \frac{t_a + t_n}{t_a M} W. \tag{6}$$

We will use these as a check on the other analyses.

IV. The Distribution of Finishing Time for One Processor

A. Introduction

We would now like to find the pdf of the finishing time for any particular algorithm or program. This is also known as the *first passage time*, the first time at which the accumulated work is greater than some particular amount (W minutes in our case). In this section, we analyze the behavior of a program on a single, transient processor using two methods. The direct method, in Section IV-B, yields $f(t)$ for general distributions, but unfortunately, it does not extend to multiple processors because the analysis depends upon the system being either fully available or fully nonavailable. In a multiprocessor system, we usually have partial availability: some of the machines are available and some are not. We do not derive the pdf of accumulated work, $y(u \mid t)$, using the direct analysis. However, by analyzing the problem as a cumulative, alternating, renewal process (Section IV-C), we do find the asymptotic probability density of the accumulated work as $t \to \infty$, and we use this in the Brownian motion analysis of the next section.

B. Direct Analysis of a Single Processor

We make a direct analysis of the single-processor problem by counting the number of nonavailable periods that interrupt our program before it completes. If our program starts at the beginning of an available period, as shown in the middle of Fig. 2, it will finish at time $W + T_a$, where T_a is the *additional* time the program spends in the system because of interrupting nonavailable periods.

Because we must finish the program in an available period, and because we assume work starts at the beginning of an available period, then none of the nonavailable periods are truncated, and it is relatively easy to analyze the total length of the nonavailable periods during the time $T_a + W$. By examining the arrival process for nonavailable periods, we find that the Laplace transform of the finishing time density is

$$F^*(s) = e^{-Ws} \sum_{k=0}^{\infty} [\mathrm{N}^*(s)]^k \, p(k \mid W)$$

$$= e^{-Ws} \mathrm{P}(\mathrm{N}^*(s)) \tag{7}$$

where $\mathrm{P}(z) = \sum_{k=0}^{\infty} p(k \mid W) z^k$ is the z-transform of $p(k \mid W)$, the probability that k zero-length nonavailable periods arrive in a W minute period starting at the beginning of an available period, and $\mathrm{N}^*(s)$ is the Laplace transform of the length of a nonavailable period. Note that we can also get the finishing time density directly using the same technique, but this usually results in an open expression; details are available in [1]. From the Laplace transform, we can derive the finishing time's mean:

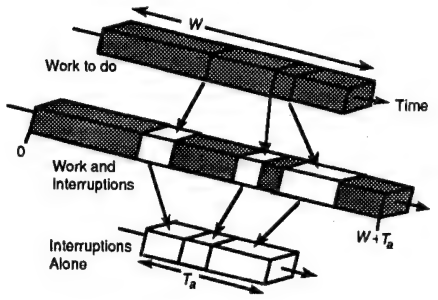


Fig. 2. Time for one node to finish W units of work.

$$\overline{f} = W \frac{t_a + t_n}{t_a}, \tag{8}$$

and variance:

$$\sigma_f^2 = \sigma_n^2 \overline{p} + t_n^2 \sigma_p^2 \tag{9}$$

where $\overline{p}$ and σ_p^2 are the mean and variance of $p(k \mid W)$.

The central limit theorem assures us that when we sum many independent random variables, the resulting distribution tends toward a normal distribution. We may note an important consequence of this: asymptotically, for large W compared to t_a and t_n, the finishing time density is normal with mean and variance given in (8) and (9).

1) Example: Exponential Distributions: If available and nonavailable periods are exponentially distributed, then the finishing time density is

$$f(t) = \begin{cases} e^{-W/t_a} & \text{if } t = W \\ \sum_{k=1}^{\infty} \left(\frac{(1/t_n)((t-W)/t_n)^{k-1}}{(k-1)!} e^{-(t-W)/t_n} \right) \left(\frac{(W/t_a)^k}{k!} e^{-W/t_a} \right) & \text{if } t > W \end{cases}. \tag{10}$$

This was derived using a direct analysis detailed in [1]. Fig. 3 illustrates this density using Mutka and Livny's parameters to select the means t_a = 91 min and t_n = 31.305 min of the exponential distributions. The mean finishing time is

$$\overline{f} = W \frac{t_a + t_n}{t_a} \tag{11}$$

and its variance is

$$\sigma_f^2 = \frac{2 t_n^2 W}{t_a}. \tag{12}$$

The finishing time density looks similar to a normal density, but it is asymmetrical. For t less than the mean first passage time, it rises sharply to a peak before the mean, then drops into a stretched-out tail for large t. This asymmetry is more apparent for small W, and as W grows, the density becomes

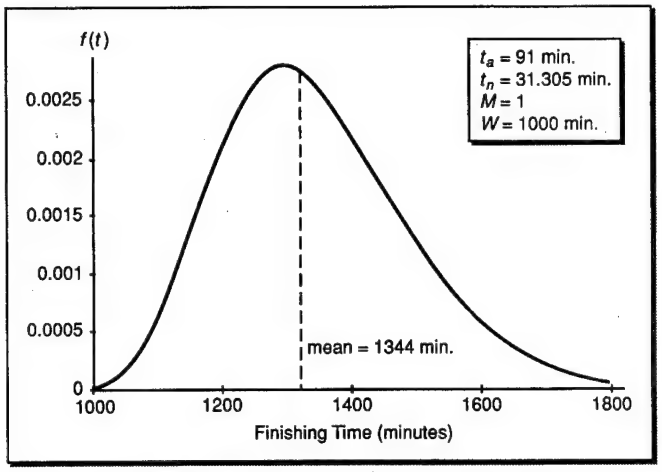


Fig. 3. Probability density of finishing time for direct analysis.

more similar to a normal density, as one would expect from the central limit theorem.

Unfortunately, the analysis of this section will not extend to multiple processors because it depends upon the system being fully available or fully nonavailable. With multiple processors the system is usually partially available. However, it is possible to get an approximation to the finishing time density for the case of multiple processors by studying a different process, namely the accumulated work. Thus, in the next section we use a cumulative, alternating renewal process to analyze the accumulated work on a single processor, then in Section V-A we apply this analysis to the multiple processor case.

C. Cumulative, Alternating Renewal Theoretic Analysis

Here we reexamine a single processor using a cumulative, alternating renewal process. Cox, in his book on renewal processes [31], discusses this type of process, and we make use of his analysis.

We can form a renewal process from the alternating states of a transient processor by letting a renewal period be a nonavailable period followed by an available period. In Fig. 4 the heavy dots indicate the beginning of each renewal period. The durations of the available periods are i.i.d. random variables from a general distribution, as are the the durations of the nonavailable periods, and the lengths of the available and nonavailable periods are mutually independent. Using Cox's results, we find that the distribution of accumulated available time has mean and variance:

$$\mathrm{E}[y(u \mid t)] \approx \frac{t_a}{t_a + t_n} t \tag{13}$$

$$\mathrm{Var}[y(u \mid t)] \approx \frac{\sigma_a^2 t_n^2 + t_a^2 \sigma_n^2}{(t_a + t_n)^3} t. \tag{14}$$

Cox derives these by ignoring the available time accumulated in the current available period if one is in progress at time t. However, as time goes to infinity, the asymptotic distribution of this approximation has the same properties as the true accumulated available time. Note that the approximation for $\mathrm{E}[y(u \mid t)]$ leads to a mean which corresponds exactly to the equilbrium approximation of (3) in Section III.

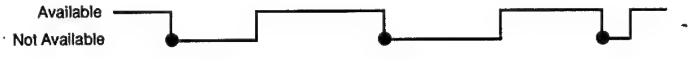


Fig. 4. Cumulative renewal process for a processor.

For exponentially distributed available and nonavailable periods, the mean remains the same and the variance may be rewritten as

$$\mathrm{Var}[y(u \mid t)] \approx \frac{2 t_a^2 t_n^2}{(t_a + t_n)^3} t. \tag{15}$$

We could use this approach to get a (possibly very complex) expression for the distribution of accumulated available time; this was also done in [1] based on the techniques of the previous section. Of more interest to us is the fact that the asymptotic pdf of the accumulated available time (for large t) is normal with mean and variance given by (13) and (14). This distribution is based on a sum of random variables (the available periods), and the Central-Limit Theorem [32] tells us that as the number of random variables in the sum approaches infinity, this pdf converges to a normal pdf. Thus the pdf of the accumulated work, $y(u \mid t)$, is well approximated by a normal pdf if many renewal periods have occurred, or equivalently, if $t \gg t_a + t_n$. This normal pdf will be the basis of the Brownian motion model in the next section, which will lead us to an approximation for the distribution of finishing time with multiple processors.

V. The Distribution of Finishing Time for M Processors

A. Brownian Motion Approximation

Brownian motion concerns the random movement of a particle through space. A stochastic process, $Q(t)$, that describes Brownian motion has two basic properties. The first is that $Q(t)$ has *independent increments*: $Q(t_1) - Q(t_0)$ and $Q(t_3) - Q(t_2)$ are independent for $0 \le t_0 < t_1 < t_2 < t_3 < \infty$. Movement of the particle in one interval is independent of its movement in another interval. The second property is that each increment in the process, $Q(t_{i+1}) - Q(t_i)$ for all $i \ge 0$, is normally distributed with a mean and variance proportional to $t_{i+1} - t_i$. If the normal distribution has mean 0 and variance equal to $t_{i+1} - t_i$, then the process describes *standard Brownian motion*, which is also known as a *Wiener process*. Brownian motion with a nonzero mean is known as *Brownian motion with drift*.

In our model, we let the stochastic process $Q(t)$ represent the amount of work accumulated by a network of M transient processors up to time t. In Section IV-C, we found that over a long period of time (much longer than $t_a + t_n$), the amount of work done by one transient processor is asymptotically normal with mean and variance given in (13) and (14), respectively. If we have a network of M such processors, and all the processors are assumed to be independent and identical, then, asymptotically, the amount of work done by time t is the sum of M independent, (approximately) normally distributed random variables, and this is itself (approximately) normally

distributed. The mean amount of work done by time t is

$$\mu = \frac{t_a}{t_a + t_n} Mt = p_a Mt \tag{16}$$

and the variance of the amount of work done by time t is

$$\sigma^2 = \frac{\sigma_a^2 t_n^2 + \sigma_n^2 t_a^2}{(t_a + t_n)^3}\ Mt. \tag{17}$$

Thus, Brownian motion with drift is a natural model of our system. From μ and σ^2 above, we define $\overline{b}$ and σ_b^2 (b signifies Brownian), the mean and variance of the amount of work accumulated per unit time:

$$\overline{b} = \frac{t_a}{t_a + t_n} M = p_a M \tag{18}$$

$$\sigma_b^2 = \frac{\sigma_a^2 t_n^2 + \sigma_n^2 t_a^2}{(t_a + t_n)^3}\ M. \tag{19}$$

We use $\overline{b}$ and σ_b^2 later in this analysis. Note that $\overline{b}$ agrees with the average accumulated work that we found in Section III.

We must still assure ourselves that our stochastic process indeed has independent increments. On a short term scale, this is clearly not true. Two consecutive one-minute intervals are likely to have the same, or at least similar, numbers of available processors in both intervals, and hence similar amounts of work accumulated in those intervals. However, in two one-minute intervals separated by several hours, the number of available processors is quite unrelated (unless the network has some very unusual statistical properties), and the work accumulated in one interval is quite independent of the work accumulated in the other interval. Thus we conclude that the Brownian motion model is reasonable only over a long span of time, and we insure this by specifying that $t_a \ll W$ and $t_n \ll W$. Note, too, that we are using the asymptotic results of Section IV-C, and these are valid only for a long span of time, which also requires a large W relative to t_a and t_n.

The Brownian motion model does allow some behavior that seemingly cannot occur in a real network. For example, the process is allowed to move in the negative direction, implying that we can lose work that we have already done. This is an artifact of the model, and it is particularly apparent at small t, but it is negligible for the conditions under which the Brownian motion model is useful. Given $\overline{b}$, σ_b^2, and t, and using the fact that cumulative work is normally distributed, we can compute the probability of negative cumulative work as

$$\Phi\left(\frac{-\overline{b}t}{\sqrt{\sigma_b^2 t}}\right) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{-(\overline{b}t/\sqrt{\sigma_b^2 t})} e^{-\frac{1}{2}x^2}\, dx \tag{20}$$

where $\Phi(x)$ is the cumulative density function for a standard normal distribution. For t near 0, $\Phi(-\overline{b}t/\sqrt{\sigma_b^2 t}) \approx 0.5$, meaning that for very small t, our model says that almost 50% of the time the program has accumulated a negative amount of work. Clearly, Brownian motion is a poor model of networks of transient processors for very small t. If we manipulate the expression $-\overline{b}t/\sqrt{\sigma_b^2 t}$ we find it is equal to $-\sqrt{(Mt_n/2(1-p_a))t}$, and the coefficient of t under the radical is greater than 1 for any reasonable values of M, t_a, and t_n. Thus as time passes and t moves away from 0 and becomes large, $-\sqrt{(Mt_n/2(1-p_a))t}$ becomes quite negative and $\Phi((-\overline{b}t/\sqrt{\sigma_b^2 t})$ shrinks to near 0. In fact, for $t = 3\sqrt{\sigma_b^2/\overline{b}}$, the 3σ point, the probability that we have negative work is approximately 0.0023 and drops rapidly thereafter to negligible amounts as t grows. This is just further confirmation that our model is valid only for relatively large W that requires more than a short time to complete.

Using $\overline{b}$ and σ_b^2 as the parameters for our Brownian motion, and using results in Karlin and Taylor [33], we find the probability density of the time, t, that it takes for M processors to finish W minutes of work is

$$f(t) = \frac{W}{\sqrt{2\pi\sigma_b^2 t^3}}\ \exp\left[-\frac{(W - \overline{b}t)^2}{2\sigma_b^2 t}\right]. \tag{21}$$

This has mean

$$\overline{f} = \frac{W}{\overline{b}} = \frac{W}{M}\frac{(t_a + t_n)}{t_a} \tag{22}$$

and variance

$$\sigma_f^2 = \frac{W}{\overline{b}}\frac{\sigma_b^2}{\overline{b}^2}. \tag{23}$$

Note that for the case $M = 1$, the mean and variance agree with the direct analysis of Section IV-B. Of course the mean is consistent with that of Section III for all M.

B. Example: Exponential Distributions

If we assume that both available and nonavailable periods are exponentially distributed, then the mean and variance of the accumulated work per unit time are:

$$\overline{b} = \frac{t_a}{t_a + t_n} M = p_a M \tag{24}$$

$$\sigma_b^2 = \frac{2(t_a t_n)^2}{(t_a + t_n)^3} M = \frac{2p_a^2(1 - p_a)M}{t_n}. \tag{25}$$

Applying this to (22) and (23) yields the mean of the finishing time

$$\overline{f} = \frac{W}{\overline{b}} = \frac{W}{M}\frac{(t_a + t_n)}{t_a} \tag{26}$$

and variance

$$\sigma_f^2 = \frac{W}{\overline{b}}\frac{\sigma_b^2}{\overline{b}^2} = \frac{2W}{M^2}\frac{t_n^2}{t_a}. \tag{27}$$

Fig. 5 shows the finishing time density for both the direct and the Brownian motion analyses with $t_a = 3600, t_n = 300, M = 1$, and $W = 10^5$. We note the good concordance between the two analyses.

When we have $M = 100$ processors, Fig. 6 shows the pdf of finishing time for various t_n with $t_a = 91$ min and $W = 10^4$ min. Using $t_a = 91$ min and $t_n = 31.305$ min (the standard multiprocessor example), we have $\overline{b} = 74.4t$ and $\sigma_b^2 = 887.2t$, which leads to $\overline{f} = 0.0134W$ and $\sigma_f^2 = 0.00215W$. Our example job of 10^4 min would take about a week to run on a single, dedicated processor. When run on a network of 100 transient processors, it would take 134.06 min, or about 2.25 h. This particular finishing time pdf is in Fig. 7, which shows the density both enlarged and plotted on a full time axis (starting at 0).

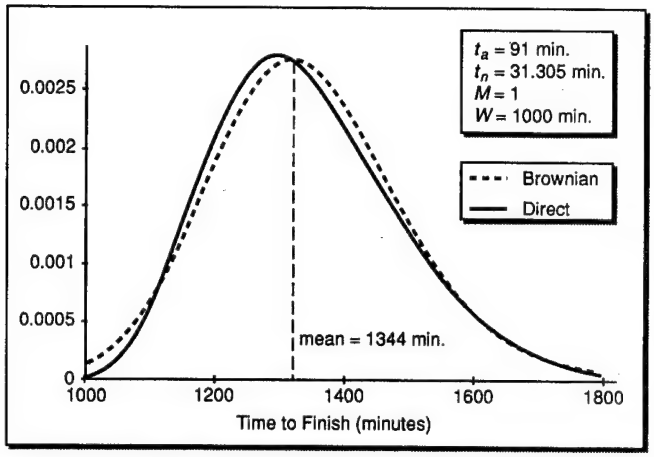


Fig. 5. Finishing time densities for direct and Brownian motion analyses.

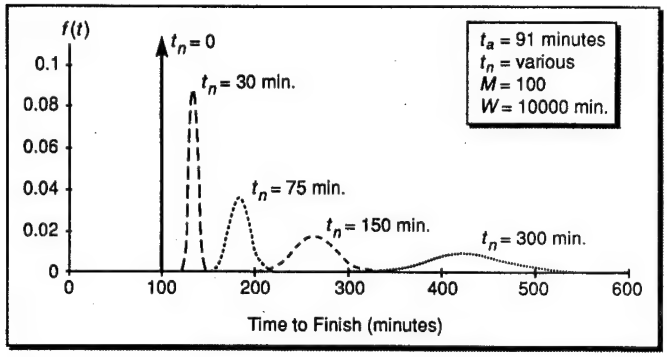


Fig. 6. Finishing time densities for Brownian motion model for various t_n.

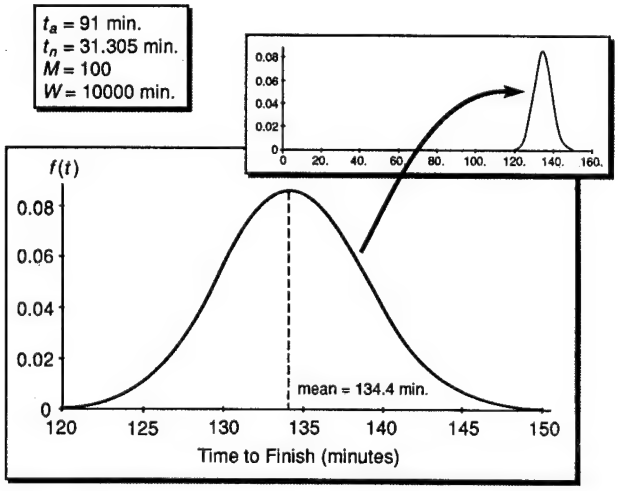


Fig. 7. Finishing time density for Brownian motion analysis.

C. Example: Mutka and Livny's Distributions

Let us use the distributions measured by Mutka and Livny. We have $t_a = 91$ min, $\sigma_a^2 = 40225$ min^2, $t_n = 31.305$ min, and $\sigma_n^2 = 2131.83$ min^2. Plugging these into (18) and (19), we find

$$\overline{b} = 74.4t \tag{28}$$

$$\sigma_b^2 = 6239t. \tag{29}$$

This leads to a finishing time mean and variance of

$$\overline{f} = 0.0134W \tag{30}$$

$$\sigma_f^2 = 0.0151W. \tag{31}$$

We note that the finishing time variance using Mutka and Livny's distributions is almost an order of magnitude more than for exponential distributions.

D. The Ratio $\sigma_f/\overline{f}$

It is instructive to examine the coefficient of variation of the finishing time, namely the ratio of σ_f to $\overline{f}$:

$$\frac{\sigma_f}{\overline{f}} = \frac{\sqrt{\sigma_b^2}}{\sqrt{\overline{b}W}}. \tag{32}$$

We note immediately that this ratio goes to zero as W increases. Consequently, for sufficiently large W, it may be accurate enough to consider the finishing time distribution as an impulse (i.e., the Dirac delta function) at the mean finishing time (in the spirit of the law of large numbers).

Assume that the available and nonavailable periods have exponential distributions. Then the ratio becomes

$$\frac{\sigma_f}{\overline{f}} = \sqrt{\frac{2t_n}{W}}\,\frac{\sqrt{t_n/t_a}}{1+t_n/t_a}. \tag{33}$$

Because we assumed $t_n \ll W$, this ratio tends to be less than 1. If we fix t_n/W and let t_n/t_a go to infinity (which implies $t_a \to 0$), the ratio goes to 0. We explain this by noting that for small t_a, it takes very many available–nonavailable cycles before the work is finished. The law of large numbers insures that the finishing time density, which is the sum of these many periods, will then be tight about its mean.

If, on the other hand, we let $t_a \to \infty$, the ratio of the standard deviation to the mean goes to zero once again. When t_a is large relative to t_n, the nonavailable periods become negligible, as if the processors are always available. Again, the finishing time density becomes very tight about its mean because nonavailable time periods add little variability to the finishing time, and under some circumstances we may consider the finishing time density as an impulse located at $t = W/M$. Using the standard multiprocessor example again ($M = 100$) with exponential distributions, we find $\sigma_f^2 = 21.53$, and approximating $f(t)$ as a normal density (discussed below), we find that 90% of the time, programs requiring 10^4 min of work will finish within 7.6 min of the 134.4 min mean finishing time, which is an interval 3% on either side of the mean. This is very narrow indeed. If we use Mutka and Livny's distributions, then 90% of the time programs finish in an interval 10 min on either side of the mean, which is still quite narrow.

We find the peak of (33) by taking the derivative with respect to t_a:

$$\frac{\partial}{\partial t_a}\frac{\sigma_f}{\overline{f}} = \frac{t_n(t_a+t_n) - 2t_at_n}{\sqrt{Wt_a}(t_a+t_n)^2}. \tag{34}$$

Setting this equal to 0 yields $t_a = t_n$ as the peak of the ratio, at which point it takes on the value $\sigma_f/\overline{f} = \sqrt{t_n/2W}$. The ratio's value at the peak is small because of our assumption that $t_n \ll W$. Note that if we assume $t_a = t_n$, but not

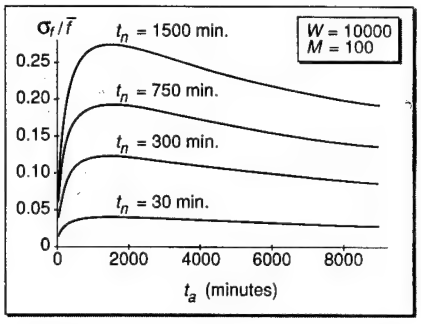


Fig. 8. $\sigma_f/\overline{f}$ with $W = 10^6$, varying t_a and t_n.

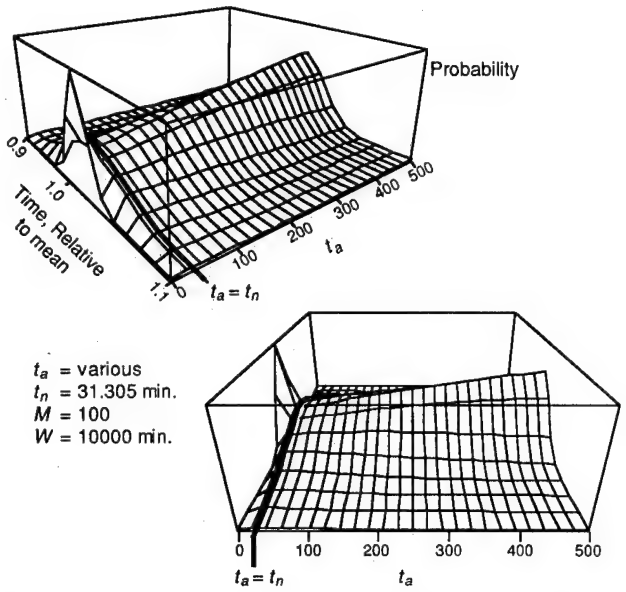


Fig. 9. Two views of the Brownian motion finishing time densities with varying t_a.

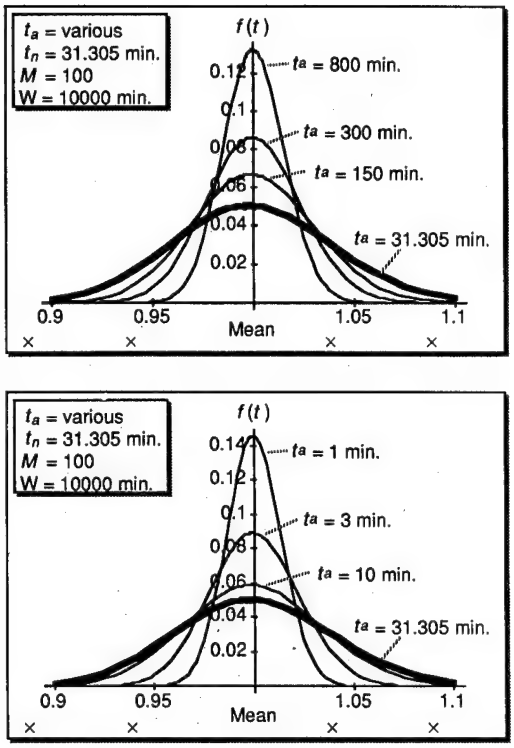


Fig. 10. Finishing time density narrowing as t_a grows (top figure) and as t_a shrinks (bottom figure).

$t_n \ll W$, then we can make the ratio as large as we want, simply by increasing t_n. If, for example, $t_a = t_n = 1$ year, then either the system is available immediately to do all our work, or else we will have to wait a very long time before it even starts. In such a case, the finishing time still has a reasonable mean but an enormous variance. Another fact to note is that M, the number of processors, does not affect the ratio $\sigma_f/\overline{f}$. Even if we have an infinite number of processors, we can still have great variance relative to the mean. Of course, both the mean and the standard deviation go to zero as M grows, but their ratio remains constant.

In Fig. 8 we plot $\sigma_f/\overline{f}$ versus t_a for $W = 10^4$, with t_n/W fixed for each curve. Fig. 9 shows finishing time densities for t_n = 31.305 min and various t_a. The x-axis (labeled "Time, Relative to Mean") is centered about the mean and plots the distance relative to the mean (varying from 0.9 times the mean to 1.1 times the mean). We note that the density is flattest and has the greatest spread for $t_a = t_n$; at this point $\sigma_f/\overline{f} = 0.035$, which is quite small. For comparison, if we use Mutka and Livny's distributions at the same point, the ratio is 0.092, which is still small. The narrowing of the density is also illustrated in Fig. 10. The parameters for both plots are: exponential distributions with varying t_a but fixed $t_n = 31.305$ min, $M = 100$, and $W = 10^4$ min. In the top plot, the density narrows as $t_a = 31.305$, 150, 300, and 800 min. In the bottom plot, we have $t_a = 31.305$, 10, 3, and 1 min as the density narrows.

E. Normal Approximation to the Finishing Time

The usual form of the central limit theorem states that the sum of n independent random variables tends to have a normal distribution as n gets large. Given this, we would expect the limiting distribution of $f(t)$ to be normal with mean $\overline{f}$ and variance σ_f^2. Let us denote this normal approximation by $\hat{f}(t)$:

$$\hat{f}(t) = \frac{1}{\sqrt{2\pi\sigma_f^2}}\, e^{-(t-\overline{f})^2/(2\sigma_f^2)}. \tag{35}$$

Substituting $t = \overline{f}$ shows that the finishing time and its normal approximation coincide at the mean:

$$f(\overline{f}) = \frac{W}{\sqrt{2\pi\sigma_b^2 W^3/\overline{b}^3}}\, e^0 = \frac{\overline{b}^{3/2}}{\sqrt{2\pi\sigma_b^2 W}}$$

$$\hat{f}(\overline{f}) = \frac{1}{\sqrt{2\pi\sigma_f^2}}\, e^0 = \frac{\overline{b}^{3/2}}{\sqrt{2\pi\sigma_b^2 W}}$$

$$= f(\overline{f}).$$

Observation shows that $f(t)$ and $\hat{f}(t)$ also coincide at two more points, but analytically these are not easily found because they are the solutions to a transcendental equation. Numer-

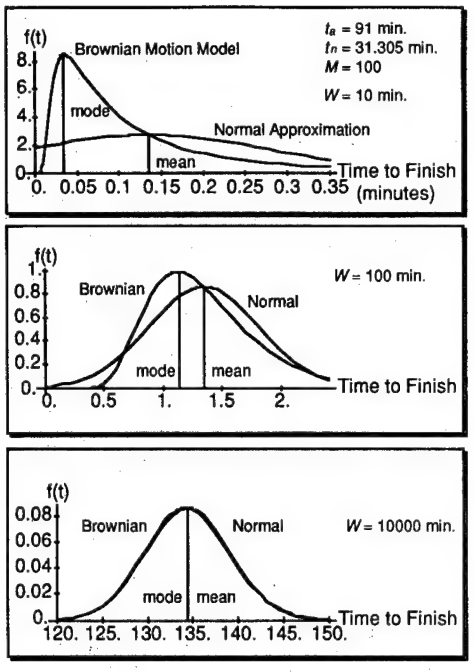


Fig. 11. Brownian motion finishing time pdf and its normal approximation.

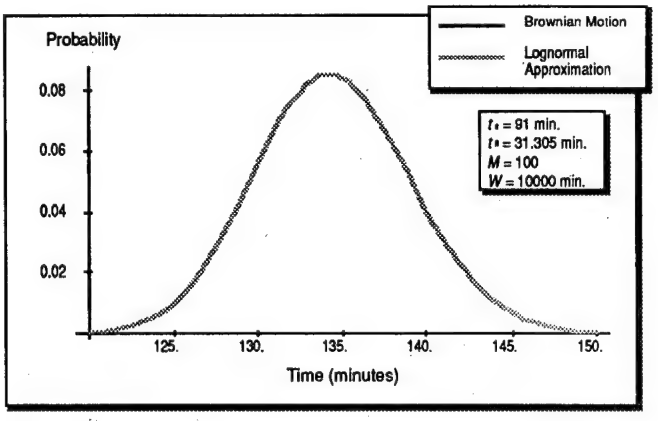


Fig. 12. Density of finishing time and its lognormal approximation.

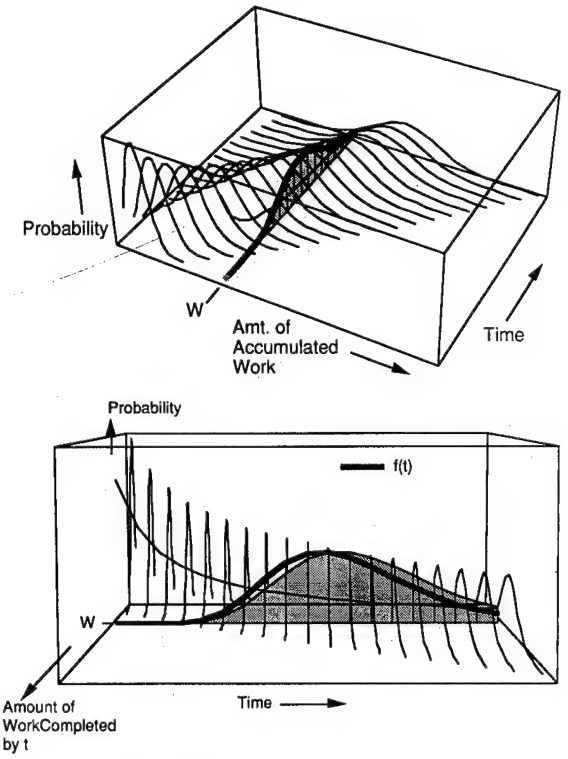


Fig. 13. Density of finishing time.

ically, we find that these points appear to be separated by $\sqrt{12\sigma_f^2}$, and the distance from the lower point (smaller t) to the mean is very slightly less than half of the total distance between the two points (varying, but in the range of 49.5% of the total separation).

We need to know when $\hat{f}(t)$ is a good approximation for $f(t)$. Observation (see Fig. 11) shows that the approximation is good when the mode of the finishing time is close to (within a few percent of) its mean. We find the mode by taking the derivative of $f(t)$ with respect to t, setting it equal to 0, and solving for t. We end up with a quadratic equation that has a negative and a positive root. The positive root is the mode, namely

$$t_{\text{mode}} = \frac{1}{2}\sqrt{\frac{9(\sigma_b^2)^2}{\overline{b}^4} + \frac{4W^2}{\overline{b}^2}} - \frac{3\sigma_b^2}{2\overline{b}^2}. \tag{36}$$

By using the fact that $\sqrt{a+b} \le \sqrt{a} + \sqrt{b}$, we show that the mode is always less than or equal to the mean:

$$t_{\text{mode}} \le \frac{3\sigma_b^2}{2\overline{b}^2} + \frac{W}{\overline{b}} - \frac{3\sigma_b^2}{2\overline{b}^2} = \frac{W}{\overline{b}} = \overline{f}.$$

Furthermore, if we observe that $9(\sigma_b^2)^2/\overline{b}^4$ is usually much less than $4W^2/\overline{b}^2$, and we use the approximation $\sqrt{1+\epsilon} \approx 1 + \frac{\epsilon}{2}$ for $0 \le \epsilon \ll 1$, then

$$\begin{aligned} t_{\text{mode}} &\approx \frac{W}{\overline{b}} - \frac{3}{2}\frac{\sigma_b^2}{\overline{b}^2}\left(1 - \frac{3}{4}\frac{\sigma_b^2}{W\overline{b}}\right) \\ &\approx \overline{f} - \frac{3}{2}\frac{\sigma_b^2}{\overline{b}^2}. \end{aligned} \tag{37}$$

Under almost all circumstances, we may drop the negative term in the parenthesis, because when the Brownian motion approximation is valid we also know that $W \gg \sigma_b/\overline{b}$, and in general, $W \gg \sigma_b$. These would render the term $3\sigma_b^2/4W\overline{b}$ negligible. Only under very unusual circumstances would $W \not\gg \sigma_b^2$, and in such cases we could not drop the term. Excepting such circumstances, (37) is quite accurate for all conditions in which our Brownian motion model is operative. Using (37), we find that the percent difference between the mean and mode is approximately $3\sigma_b/2\overline{b}W$.

F. Lognormal Approximation to Finishing Time

A lognormal density provides a remarkably good approximation to (21). A lognormal distribution has two parameters, μ_l and σ_l^2. If we equate the mean and variance of the finishing time pdf from the Brownian motion model to that of a

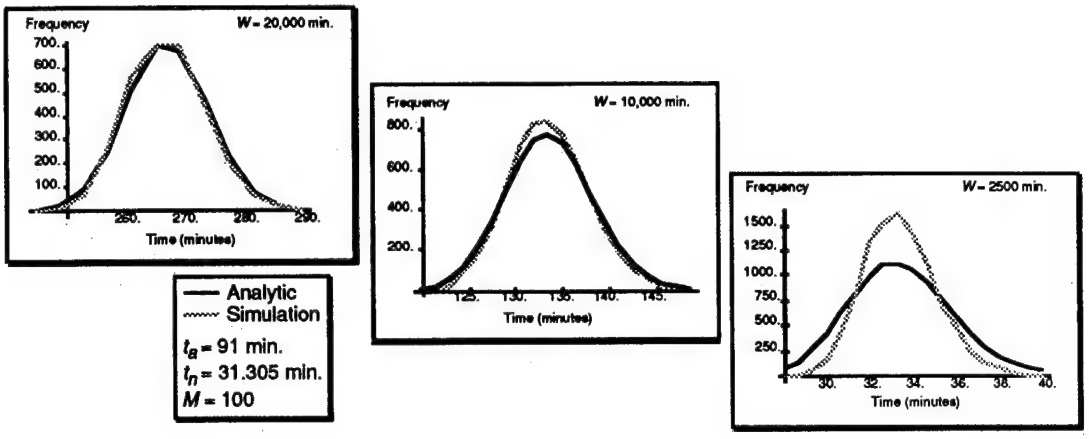


Fig. 14. Simulation results.

lognormal pdf, then we find that these parameters must be

$$\mu_l = \ln(\overline{f}) - \frac{1}{2}\sigma_l^2$$

$$= \ln\left(\frac{\overline{f}}{\sqrt{\sigma_f/\overline{f}^2 + 1}}\right) \tag{38}$$

$$\sigma_l^2 = \ln(\sigma_f^2/\overline{f}^2 + 1). \tag{39}$$

The lognormal pdf fit to (21) then becomes

$$l(t) = \frac{1}{t\sqrt{2\pi\sigma_l^2}} \exp\left[-\frac{(\ln(t) - \mu_l)^2}{2\sigma_l^2}\right]. \tag{40}$$

As shown in Fig. 12, when both the Brownian motion finishing time pdf and the lognormal approximation are plotted, the densities are extremely close, and the plotted curves appear to lie on top of each other. When the two do differ, it is under circumstances where the assumptions of the Brownian motion model do not hold (e.g. W small relative to all of M, t_a, and t_n).

G. The Finishing Time Density and its Derivation from a Normal pdf

We can rewrite the Brownian motion finishing time density, (21), in a form using a normal pdf. Let $\phi_{\mu,\sigma^2}(x)$ be the probability density that a random variable, normally distributed with mean μ and variance σ^2, takes on the value x. Using this, (21) becomes

$$f(t) = \frac{W}{t}\phi_{\overline{b}t,\sigma_b^2 t}(W). \tag{41}$$

The normal pdf term derives from the underlying Brownian motion; it is the probability that a total of W units of work have been accumulated by time t. As for the weighting factor of W/t, no intuitive explanation has yet been found for this. Fig. 13 illustrates the relationship between (21) and (41). In this figure, the normal pdf of the amount of work done by time x, $\phi_{\overline{b}t,\sigma_b^2 t}(x)$, is plotted with thin lines for various t. The shaded plane in the figure picks out those points on the normal pdf where $x = W$; the line arcing down (top left to lower right in the bottom view) within this plane represents W/t. The other line within the shaded plane is the finishing time density, i.e., the product of these last two curves. The curves have been scaled differently to make them fit into one plot, so relative heights, except within the group of normal curves, are meaningless.

H. Simulation Results

We ran simulations for the case of exponentially distributed available and nonavailable periods. Some results comparing the simulation to the Brownian motion model are shown in Fig. 14. The Brownian motion model and the simulation agree very well for large W, as we would expect, and they deviate as W becomes small.

VI. Conclusion

In this paper, we analyzed the distribution of the time to finish a distributed program running in a network of transient processors. We first made two analyses of a program running on a single transient processor. These results were then used as the basis for a Brownian motion, multiprocessor model, and from this we found four finishing time distributions: the actual Brownian motion finishing time distribution, and its normal, lognormal, and impulse approximations. The models in this paper offer an approach to predicting performance of distributed programs on transient processors. By relaxing some of our assumptions, as discussed below, more sophisticated models could be derived from those that have been described.

The first assumption that we would like to relax concerns the asymptotic nature of the results. The results we have given are valid only over a long period of time. If we have a relatively small amount of work to do (say, several hours), then our finishing time distributions are not valid. Their means are acceptable, but the variance is quite incorrect, and the distributions (except for the impulse approximation) show noticeable probability that the program will finish in less than the minimal time required (W/M). Judging from the simulation results, there may well be some simple way to heuristically modify the variance expression in our models so they provide acceptable results for small W.

A second assumption we would like to relax concerns our model of a program. Modeling an algorithm as sequential, independent stages is very simplistic. Many programs do not have clear stages, but instead have a more complex internal precedence structure among the tasks of the program which cause additional delays. The independent-stages model may provide a useful simplification, but testing this, and developing more complicated program models, remains for future work.

A third assumption of great importance is that our network model does not account for the realities of communications. Communication entails delay, and our model does not address this issue. Solutions to this are currently under investigation, and some possibilities are mentioned in [1].

The models in this paper do have many assumptions, yet their very simplicity makes them appealing. An alternative to our models is performability analysis, yet the complexities of performability models yield only numeric results or a Laplace transform, and not a direct analytic expression for the distribution of program completion time. Furthermore, the basic parameters of our models can be modified to remove some of the assumptions, and, at least for large W, the models already do capture the basic behavior of transient, distributed systems.

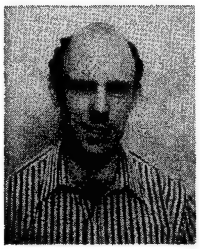
ACKNOWLEDGMENT

The authors would like to thank the referees for their very useful suggestions.

REFERENCES

[1] W. Korfhage, "Distributed systems and transient processors," Ph.D. dissertation, Univ. California, Los Angeles, Aug. 1989.
[2] L. Kleinrock, "Distributed systems," *Commun. ACM,* vol. 28, pp. 1200–1213, Nov. 1985.
[3] S. J. Mullender *et al.*, "Amoeba: A distributed operating system for the 1990s," *IEEE Comput. Mag.,* vol. 23, pp. 44–53, May 1990.
[4] M. W. Mutka and M. Livny, "Scheduling remote processing capacity in a workstation-processor bank network," in *Proc. 7th Int. Conf. Distributed Comput. Syst.,* Berlin, Germany, Sept. 1987.
[5] P. Kreuger and R. Chawla, "The stealth distributed scheduler," in *Proc. 11th Int. Conf. Distributed Comput. Syst.,* IEEE Computer Society, May 1991, pp. 336–343.
[6] R. Alonso and L. L. Cova, "Sharing jobs among independently owned processors," in *Proc. 8th Int. Conf. Distributed Comput. Syst.,* June 1988, pp. 282–288.
[7] R. Felderman, E. Schooler, and L. Kleinrock, "The benevolent bandit laboratory: A testbed for distributed algorithms," *IEEE J. Select. Areas Commun.,* vol. 7, pp. 303–311, Feb. 1989.
[8] J. R. Lyle and C. Lu, "Load balancing from a unix shell," in *Proc. 13th Conf. Local Comput. Networks,* Oct. 1988, pp. 181–183.
[9] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor - A hunter of idle workstations," in *Proc. 8th Conf. Distributed Comput. Syst.,* San Jose, CA, June 1988.
[10] M. W. Mutka and M. Livny, "Profiling workstation's available capacity for remote execution," Computer Sciences Tech. Rep. 697, CS Dept., Univ. Wisconsin, May 1987.
[11] D. A. Nichols, "Using idle workstations in a shared computing environment," in *Proc. Eleventh ACM Symp. Oper. Syst. Principles,* ACM, Nov. 1987, pp. 5–12.
[12] M. Stumm, "The design and implementation of a decentralized scheduling facility for a workstation cluster," in *Proc. 2nd IEEE Conf. Comput. Workstations,* Mar. 1988, pp. 12–22.
[13] J. F. Shoch and J. A. Hupp, "The "worm" programs — Early experience with a distributed computation," *Commun. ACM,* vol. 25, pp. 172–180, Mar. 1982.
[14] T. J. Shimeall, personal communication, 1989.
[15] B. T. Doshi, "Queueing systems with vacations — A survey," *Queueing Systems,* vol. 1, pp. 29–67, June 1986.
[16] H. White and L. S. Christie, "Queueing with preemptive priorities or with breakdown," *Oper. Res.,* vol. 6, pp. 79–95, Jan.–Feb. 1958.
[17] K. Thiruvengadam, "Queueing with breakdowns," *Oper. Res.,* vol. 11, pp. 62–71, Jan.–Feb. 1963.
[18] D. P. Gaver, Jr., "A waiting line with interrupted service, including priorities," *J. Roy. Statist. Soc.,* vol. B24, pp. 73–90, 1962.
[19] A. Federgruen and L. Green, "Queueing systems with service interruptions," *Oper. Res.,* vol. 34, pp. 752–768, Sept.–Oct. 1986.
[20] M. D. Beaudry, "Performance-related reliability measures for computing systems," *IEEE Trans. Comput.,* vol. C-27, pp. 540–547, June 1978.
[21] J. F. Meyer, "On evaluating the performability of degradable computing systems," *IEEE Trans. Comput.,* vol. C-29, pp. 720–731, Aug. 1980.
[22] J. F. Meyer, "Closed-form solutions of performability," *IEEE Trans. Comput.,* vol. C-31, pp. 648–657, July 1982.
[23] B. R. Iyer, "Recent results in performability analysis," in *Current Advances in Distributed Computing and Communications,* Computer Science Press, 1987, pp. 50–64.
[24] S. M. R. Islam and H. H. Ammar, "Performability of the hypercube (reliability)," *IEEE Trans. Reliability,* vol. 38, pp. 518–526, Dec. 1989.
[25] K. R. Pattipati and S. A. Shah, "On the computational aspects of performability models of fault-tolerant computer systems," *IEEE Trans. Comput.,* vol. 39, pp. 832–836, June 1990.
[26] B. R. Iyer, L. Donatiello, and P. Heidelberger, "Analysis of performability for stochastic models of fault-tolerant systems," *IEEE Trans. Comput.,* vol. C-35, pp. 902–907, Oct. 1986.
[27] V. G. Kulkarni, V. F. Nicola, R. M. Smith, and K. S. Trivedi, "Numerical evaluation of performability and job completion time in repairable fault-tolerant systems," in *Fault-Tolerant Computing Systems 16,* 1986, pp. 252–257.
[28] P. S. Puri, "A method for studying the integral functionals of stochastic processes with applications: I. The Markov chain case," *J. Appl. Probability,* vol. 8, pp. 331–343, 1971.
[29] R. M. Smith, K. S. Trivedi, and A. V. Ramesh, "Performability analysis: Measures, an algorithm, and a case study," *IEEE Trans. Comput.,* vol. 37, pp. 406–417, Apr. 1988.
[30] E. de Souza e Silva and H. R. Gail, "Calculating availability and performability measures of repairable computer systems using randomization," *J. ACM,* vol. 36, pp. 171–193, Jan. 1989.
[31] D. R. Cox, *Renewal Theory.* London: Methuen and Co., Ltd., science paperbacks ed., 1962.
[32] A. Mood, F. Graybill, and D. Boes, *Introduction to the Theory of Statistics.* New York: Series in Probability and Statistics, McGraw-Hill, 1974.
[33] S. Karlin and H. M. Taylor, *A First Course in Stochastic Processes,* second ed. New York: Academic, 1975.

Leonard Kleinrock (S'55–M'64–SM'71–F'73), for a photograph and biography, see the March 1993 issue of the TRANSACTIONS, p. 317.



Willard Korfhage (S'81–M'89) received the B.S. degree in electrical engineering and computer science from Princeton University in 1982, and the M.S. and Ph.D degrees in computer science from U.C.L.A in 1985 and 1989, respectively.

He is an assistant professor of Computer Science at Polytechnic University, Brooklyn, NY. He joined the faculty at Polytechnic in 1989, where he is a member of the Center for Advanced Technology in Telecommunication and runs Polytechnic's Distributed Systems Laboratory. His research interests are in distributed systems. He and his students are putting process migration and load balancing in the Hermes language, and creating Panorama, a distributed monitoring and visualization system.

packets from the Convergence Sublayer – Protocol Data Unit (CS-PDU). Transport layer protocols will need to be designed and optimized to support these very large super-packets. North Carolina State University, under contract with MCNC, is currently doing research in this area to quantify the performance of various alternatives for these transport layer protocols. OA&M is an important part of any large network. The VISTAnet project will provide hands-on experience in managing a B-ISDN network. From this experience, the OA&M capabilities required to support B-ISDN can be anticipated and developed prior to its deployment in the 1994–1995 time frame. End-to-end flow control will be required for some services in order to prevent the overflow of buffers in the terminal equipment. Such flow control capabilities are provided between the HIPPI NTAs of the VISTAnet project. In this case, Ready cells carry information about the current buffer availability in the destination NTA. Research in this area will study the effectiveness of this approach.

B-ISDN congestion control issues are now under urgent study in CCITT Study Group XVIII. By switching actual traffic through the ATM switch in the VISTAnet project, the effects of congestion on an ATM cell stream can be characterized, and the effectiveness of specific congestion control alternatives can be studied. B-ISDN must make effective use of various technologies in providing its switching capabilities, and these technologies must interwork effectively. VISTAnet will enable the study of the interworking of circuit switching and ATM switching technologies in the same testbed.

9. Conclusions

We expect that the VISTAnet testbed will enable important advances in communications research, computer visualization, and radiation therapy planning. This project should demonstrate the suitability of a public broadband network for even the most advanced applications. After the network becomes fully operational, extensions of the network to additional research teams are anticipated to study more complex traffic. In particular, extensions to other gigabit trial networks may be considered.

References

[1] D. Beard, "Designing a Radiology Workstation: A Focus on Navigation During the Interpretation Task", *Journal of Digital Imaging* **3** (3), 152–163.

[2] American National Standard for Information Systems, *High-Performance Parallel Interface – Mechanical, Electrical, and Signaling Protocol Specification*, X3T9.3/88-023 Rev. 7.2, (Aug. 1, 1990).

[3] Y. Kato, T. Shimoe, and K. Murakami, "Development of a High Speed ATM Switching LSIC", in *Proceedings ICC '90*, 1990.

[4] J. Rosenman, "Dynamic Radiation Therapy Treatment Planning: An Application of Gigabit Networking", in *The 2nd Annual Workshop of Very High Speed Networks*, Mar. 4–5, 1991.

[5] D. R. Spears, "Broadband ISDN – Service Visions and Technological Realities", *International Journal of Digital and Analog Cabled Systems* **1** (1), (1988).

Journal of High Speed Networks 1 (1992), 61–77
IOS Press

PERFORMANCE ANALYSIS OF SINGLE-HOP WAVELENGTH DIVISION MULTIPLE ACCESS NETWORKS*

Jonathan C. Lu and Leonard Kleinrock
Computer Science Department
University of California, Los Angeles
3732L Boelter Hall
Los Angeles, CA 90024-1596, USA
lu@cs.ucla.edu, lk@cs.ucla.edu

Abstract

Wavelength division multiple access (WDMA) provides a way to tap the huge bandwidth of an optical fiber by simultaneously operating on multiple channels at different wavelengths, with each channel running at the speed of the electronics of an end user station. This paper presents a mathematical model which approximates WDMA networks with general hardware configurations and arbitrary traffic patterns. Packets which cannot be transmitted upon arrival are blocked (i.e., lost) immediately. We first study the case of a uniform traffic matrix and observe that, when the number of wavelengths is fewer than the number of stations, it is better to have both tunable transmitters and tunable receivers, rather than having only either one of them tunable. Furthermore, we find that only a small number of tunable transmitters and receivers per station is needed to produce performance close to the upper bound. We then construct a general traffic model and propose an iterative solution procedure. A case of hot-spot traffic is studied using this model. We find that adding more resources to the hot-spot node will help improve its performance, but only to a limited extent determined by the traffic imbalance. The match between the model and simulation results are shown to be excellent.

Keywords: Wavelength Division Multiplexing, Performance Analysis, Fiber Optics

1. Introduction

The rapid development of lightwave technology offers the potential of a huge amount of bandwidth in a single optical fiber. It is conceivable that we could construct multiple access networks with a total capacity of around 50 terabits per

* This work was supported by the Defense Advanced Research Projects Agency under Contract MDA 903-87-C0663, Parallel Systems Laboratory.

second by using the low-loss passband of optical fibers (1200–1600 nm) [1]. An obstacle to realizing such high-capacity networks lies in the bottleneck at the electronic interface, which can modulate/demodulate the light at a mere fraction of the optical bandwidth. Therefore, to tap the bandwidth potential of optical fibers, the network architecture must employ some form of concurrency, i.e., the ability to simultaneously convey a multitude of distinguishable messages. One such approach, called Wavelength Division Multiple Access (WDMA), could achieve this by operating on multiple channels at different wavelengths, with each channel running at the speed of the electronics of an end user station. By assembling a large number of wavelength-multiplexed channels, WDMA carries the potential of providing the network capacity required by future applications.

One class of WDMA networks is the multi-hop network [2, 3], which is constructed by setting the transmitters and/or receivers of a station to be tuned at certain fixed wavelengths. A link is formed between two nodes when a transmitter of one node and a receiver of the other node both tune to the same wavelength. The way these transmitters and receivers are tuned defines an interconnection pattern. A packet may be routed through several intermediate nodes before it is delivered to its destination. An early proposal for the interconnection pattern consisited of several stages connected through a Perfect Shuffle [4]. However, there is no a *priori* reason to be restricted to this interconnection pattern. Two other papers [5, 6] propose schemes to optimize the logical connectivity by (slowly) retuning the transmitters and receivers of the stations adaptively to the traffic. Another class of WDMA networks [7–10] assumes single-hop communications which employs tunable transmitters and/or receivers with rapid tuning to dynamically set up connections between stations on a per packet basis. Both the single-hop and multi-hop networks can achieve an aggregate throughput substantially larger than the electronic speed of a single station. An advantage of single-hop over multi-hop communications is that multi-hop implies longer routes and thus larger propagation delays, which is the dominating delay component in high-speed networks. In this paper we consider only single-hop cases.

Ramaswami and Pankaj [11] compared having either tunable transmitters only, or tunable receivers only, or both, assuming each station is equipped with only one transmitter and one receiver. Chlamtac and Ganz [12] discussed the design alternatives of WDMA star networks where each station can have multiple transmitters and receivers and some finite buffers. Both of these two previous studies were conducted only for the case of a uniform traffic matrix. The purpose of this paper is to present a mathematical model for WDMA networks to examine the effects of resource contention (of transmitters, receivers, and wavelengths) under general traffic patterns. Each station may have its own hardware configuration and traffic requirement. Our model ignores any specific media access protocol by assuming that each station has perfect knowledge of the current status of all the resources in the system. This assumption is reasonably good for the case of a packet switch where the physical distance is small and stations can learn the status of the resources from information broadcast by a centralized controller. The model serves as an upper bound on performance when the system is a network which covers a larger geographical area.

The rest of the paper is organized as follows. In Section 2, we describe the system configuration and assumptions to be used in the mathematical model. Section 3 presents the analysis of networks with stations having multiple transmitters and receivers for the uniform traffic case. A general model is constructed in Section 4 and an iterative procedure is proposed to solve it for the general traffic case. In Section 5, a hot-spot traffic case is then studied using the general model. Section 6 gives the conclusions.

2. The System Model and the Solution Method

The system considered here consists of N stations attached to a broadcast medium (fiber bus or star coupler). The number of wavelengths is equal to W. Node[1] i has t_i transmitters and r_i receivers, each of which may be tunable to any wavelength or which may be tuned to a single fixed wavelength. We assume that a stream of packets arrive to node i following a Poisson process with rate λ_i packets per unit time. The packet length is exponentially distributed with mean $1/\mu$, the same for all nodes. We shall choose the average packet length as the time unit of the system by setting $\mu = 1$ throughout the analysis. A packet arriving at node i is addressed to destination node j with probability x_{ij}, $1 \le i, j \le N$. Define

$$\phi_i \triangleq \sum_{j=1}^{N} \lambda_j x_{ji} \qquad 1 \le i \le N$$

as the intensity of generated traffic that is destined for node i. For a packet to be transmitted and successfully received, the three following conditions must all be satisfied simultaneously: (i) there is a free wavelength in the system, (ii) there is a free transmitter, at the source node, which can access that free wavelength, and (iii) there is a free receiver, at the destination node, which can also access that same free wavelength. We assume there is no buffering at each node. Upon a packet's arrival, it is transmitted immediately if all the three conditions above are true (remember that we have assumed a "perfect" access scheme); otherwise the packet is blocked (i.e., lost) immediately. We assume that each station has complete knowledge of the status (busy or idle) of all the wavelengths, transmitters, and receivers in the system. The throughput of the system, which is defined as the average number of successful packets transmitted per unit time, will be used as the performance measure to compare systems with different configurations and different traffic patterns.

Let the random variable K be the number of busy wavelengths in the system in steady state. Let $p_k \triangleq Prob[K = k]$, $0 \le k \le W$. Knowing the number of busy wavelengths does not completely describe the state of the system since we also need the current status of the transmitters and receivers of each node. However, we will make the approximation that K is a Markov chain. In this analysis, we will also approximate many of the transition rates of this chain and then provide an exact solution under these approximations. Given that the system is in state k, $0 \le k \le W-1$, and given a specific free wavelength, we define $\alpha_k^{(i)}$ as the

[1] The words ***node*** and ***station*** will be used interchangeably throughout this paper.

probability that an arriving packet at node i finds at least one of its transmitters free which can access that free wavelength, and $\beta_k^{(j)}$ as the probability that a packet destined for node j arriving at a source node finds, upon its arrival, a free receiver at node j which can access that same free wavelength. We recognize that these two probabilities should properly be computed as a joint probability; we choose to approximate them by assuming independence of the underlying events. Let σ_k denote the transition rate from state k to state $k+1$ due to the transmission of a new packet. We first note that $\lambda_i x_{ij}$ is the rate of new packets generated by node i and addressed to node j. The probability that this new packet is successfully transmitted is approximately equal to $\alpha_k^{(i)}\beta_k^{(j)}$. Therefore, under the assumption that all the free wavelengths are equally favored for the transmission of a new packet, σ_k can be calculated as follows:

$$\sigma_k = \sum_{i=1}^{N}\sum_{j=1}^{N} \lambda_i x_{ij} \alpha_k^{(i)} \beta_k^{(j)} \qquad 0 \le k \le W \tag{1}$$

We see that the evolution of K forms a Markov chain which is a birth-death process whose state transition diagram is shown in Fig. 1. Solving this birth-death process [13], we have

$$p_k = p_0 \prod_{i=0}^{k-1} \frac{\sigma_i}{(i+1)\mu} \tag{2}$$

where

$$p_0 = \left[1 + \sum_{k=1}^{W}\prod_{i=0}^{k-1} \frac{\sigma_i}{(i+1)\mu}\right]^{-1} \tag{3}$$

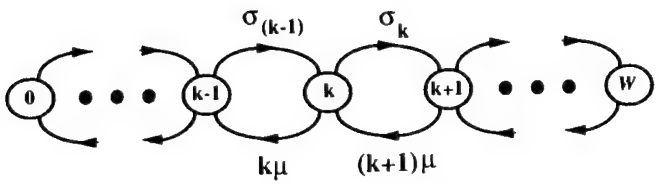


Fig. 1. State transition diagram for number of busy wavelengths in the system

The throughput of the system, S which is also equal to the average number of busy wavelengths in the system, can be calculated by

$$S = \sum_{k=0}^{W} k p_k \tag{4}$$

This, then, is the general setup for our solution. It remains to find σ_k and hence S. This we do in the next two sections.

3. The Uniform Traffic Case

In this section we study the uniform traffic case where packets arrive to a station following a Poisson process with rate λ packets per unit time (the same for all stations). A packet will travel from its source station to any of the N stations (including the source itself) with equal probability, i.e., $x_{ij} = 1/N$, $1 \le i, j \le N$. (Setting $x_{ij} = 1/(N-1)$, $1 \le i, j \le N$, $i \ne j$ does not change the results below.)

3.1. *Tunable Transmitters and Receivers*

Here we consider the case where each node is equipped with q ($q \le W$) tunable transmitters and q tunable receivers, each of which can tune to any of the W wavelengths. The $\alpha_k^{(i)}$'s and $\beta_k^{(i)}$'s are now the same for all stations by symmetry, which we denote by α_k and β_k, respectively. To get the α_k and β_k, we first note, given that the system is in state k, that it is implied that there are also k transmitters and k receivers currently busy in the system. When $k < q$, α_k (β_k) is equal to one because there must be always a free transmitter (receiver) at any node. For the cases $k \ge q$, since there is a total of Nq transmitters (receivers) in the system, we know that the probability that any single transmitter (receiver) is busy equals k/Nq. Therefore, the probability that all the transmitters (receivers) of a given node are busy is approximately equal to $(k/Nq)^q$. One minus this gives us α_k (β_k). Thus, we have the following approximation:

$$\alpha_k = \beta_k = \begin{cases} 1 & 0 \le k \le q-1 \\ 1 - \left(\frac{k}{Nq}\right)^q & q \le k \le W-1 \end{cases} \tag{5}$$

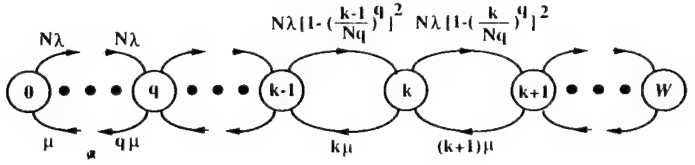


Fig. 2. State transition diagram for tunable transmitters and receivers

The transition rates σ_k can be calculated using (1) and (5), and the corresponding state transition diagram is shown in Fig. 2. Solving this Markov chain, we get

$$p_k = p_0 \frac{(N\rho)^k}{k!} \qquad 0 \le k \le q$$

$$p_k = p_0 \frac{(N\rho)^k}{k!} \prod_{i=q}^{k-1} \left[1 - \left(\frac{i}{Nq}\right)^q\right]^2 \qquad q+1 \le k \le W$$

where $\rho = \lambda/\mu$ and

$$p_0 = \left[\sum_{k=0}^{q} \frac{(N\rho)^k}{k!} + \sum_{k=q+1}^{W} \frac{(N\rho)^k}{k!} \prod_{i=q}^{k-1} \left(1 - \left(\frac{i}{Nq}\right)^q\right)^2\right]^{-1}$$

The throughput S can be calculated from (4).

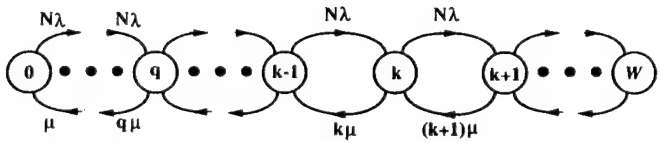


Fig. 3. An upper bound, the W-server loss system

An *achievable* upper bound on the throughput can be obtained by assuming all nodes have W tunable transmitters and receivers. In this case, $\alpha_k = \beta_k = 1$, which corresponds to a W-server loss system [13] where each wavelength corresponds to a server. Figure 3 shows the corresponding state transition diagram. Solving this, we have

$$p_k = p_0 \frac{(N\rho)^k}{k!} \qquad 0 \le k \le W$$

where

$$p_0 = \left[\sum_{k=0}^{W} \frac{(N\rho)^k}{k!}\right]^{-1}$$

The blocking probability of this upper bound system equals

$$p_W = \frac{(N\rho)^W/W!}{\sum_{k=0}^{W}(N\rho)^k/k!}$$

which is the well-known *Erlang B formula* [13].

In Fig. 4 and 5 we plot the throughput versus the total offered load for $N = 50$, $W = 10$ and $N = 50$, $W = 50$, respectively. We show the ideal upper bound on throughput as equal to the input load up to the point where the load equals the

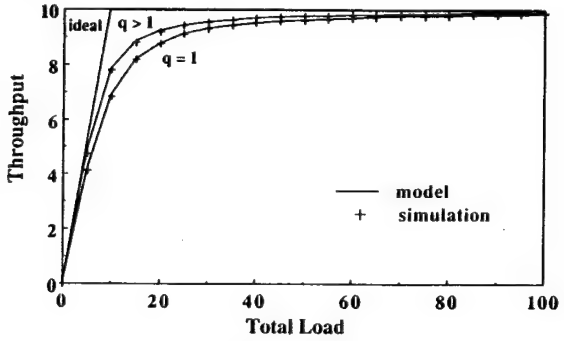


Fig. 4. Throughput versus total load $N\rho$. $N = 50$, $W = 10$

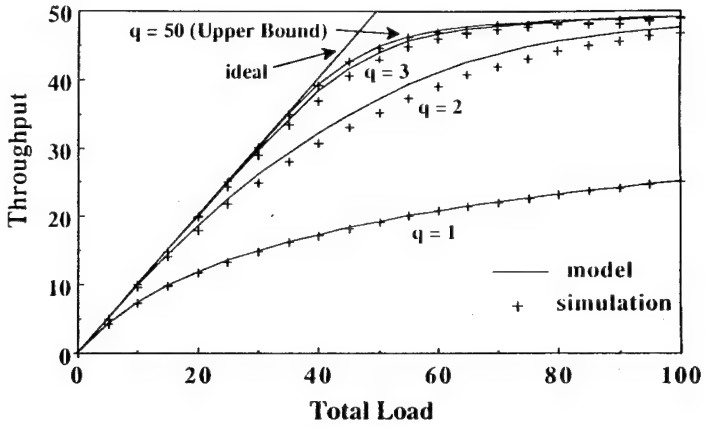


Fig. 5. Throughput versus total load $N\rho$. $N = 50$, $W = 50$

total system bandwidth; beyond that point, any additional traffic is clearly lost. We can see that a small q (much smaller than W) is enough to produce a result close to the achievable upper bound where $q = W$. This is because, in the uniform traffic case, the probability that more than a few packets are going to the same destination at the same time is very small, and only a small number of transmitters and receivers are required at each node.

3.2. *Tunability on One Side Only*

In this section we consider the same uniform traffic case except that each station now has only tunable transmitters or receivers, but not both. We begin with the case where each node is equipped with one tunable transmitter and f ($f \le W$) fixed tuned receivers. Each receiver in a station is tuned to a different fixed wavelength and the receivers in the whole system are tuned in a uniform way such that the number of recievers tuned to each wavelength is the same, which equals Nf/W (assumed to be an integer).

By the same arguments as in the previous subsection, α_k can be easily (but approximately) derived from (5) by setting $q = 1$.

$$\alpha_k = 1 - \frac{k}{N} \qquad 0 \le k \le W - 1$$

To get β_k requires a bit of different reasoning. For $k < f$, β_k equals one because the total number of busy receivers in the sytem is fewer than the number of receivers each station has. To transmit a new packet, the source node can just tune its transmitter to the free wavelength of any idle receiver at the destination. For the case $k \ge f$, recall that all the receivers are tuned in a uniform way over all the wavelengths; therefore, we know that, given that the system is in state k (i.e., there

are currently k busy wavelengths), the probability that the fixed wavelength at an arbitrary receiver at the destination is busy equals k/W. The probability that wavelengths at the receivers of a given node are all busy is approximately $(k/W)^f$, and one minus this gives us β_k as follows:

$$\beta_k = \begin{cases} 1 & 0 \le k \le f-1 \\ 1-(\frac{k}{W})^f & f \le k \le W-1 \end{cases}$$

By switching the roles of transmitters and receivers in the discussion above, we can easily obtain the α_k and β_k for the case of multiple fixed transmitters and one tunable receiver per node, which are equal to the β_k and α_k listed above, respectively; the two systems are "duals" of each other. Therefore, the state transition diagrams of those two cases are exactly the same and is shown in Fig. 6. Solving this Markov chain under our approximation, we have

$$p_k = p_0 \frac{(N\rho)^k}{k!} \prod_{i=0}^{k-1} \left(1-\frac{i}{N}\right) \qquad 1 \le k \le f$$

$$p_k = p_0 \frac{(N\rho)^k}{k!} \left[\prod_{i=0}^{k-1}\left(1-\frac{i}{N}\right)\right]\left[\prod_{i=f}^{k-1}\left(1-\left(\frac{i}{W}\right)^f\right)\right] \qquad f+1 \le k \le W$$

where $\rho = \lambda/\mu$, and

$$p_0 = \left[1 + \sum_{k=1}^{f} \frac{(N\rho)^k}{k!} \prod_{i=0}^{k-1}\left(1-\frac{i}{N}\right) + \sum_{k=f+1}^{W} \frac{(N\rho)^k}{k!}\left[\prod_{i=0}^{k-1}\left(1-\frac{i}{N}\right)\right]\left[\prod_{i=f}^{k-1}\left(1-\left(\frac{i}{W}\right)^f\right)\right]\right]^{-1}$$

The throughput can be calculated from (4).

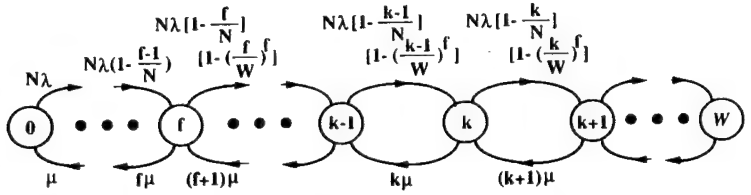


Fig. 6. State transition diagram for tunability on one side only

Figure 7 shows the case in which the number of wavelengths is small ($W = 10$) compared to the number of nodes ($N = 50$) in the system. We see that there is an interval in the light load range where multiple fixed receivers is better than one tunable receiver because not many wavelengths are in use and a station with

multiple receivers can receive more than one packet at a time. However, as the load increases the average number of wavelengths in use increases too, and it is better to have a tunable receiver than multiple fixed receivers because the wavelengths those fixed receivers are tuned to may be all in use (by other stations) and a given station could not receive any packet even though not all of its receivers were busy. Figure 8 shows the case where $N = 50$ and $W = 25$ on a different scale. Once again we see the importance of going to a single tunable receiver at heavy load. When the number of wavelengths becomes the same as the number of nodes in the

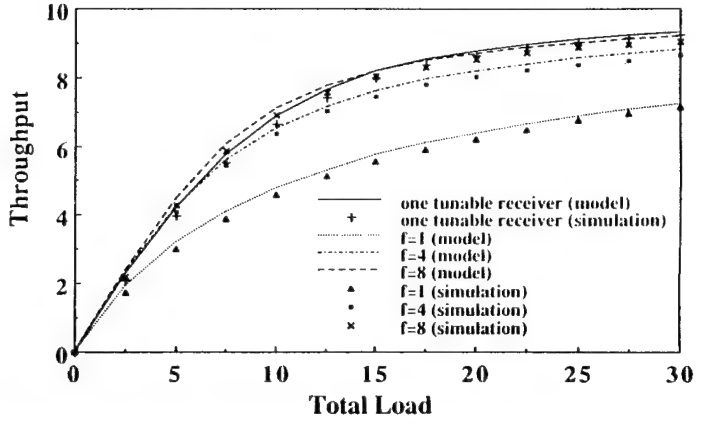


Fig. 7. Throughput versus total load $N\rho$. $N = 50$, $W = 10$, and one tunable transmitter

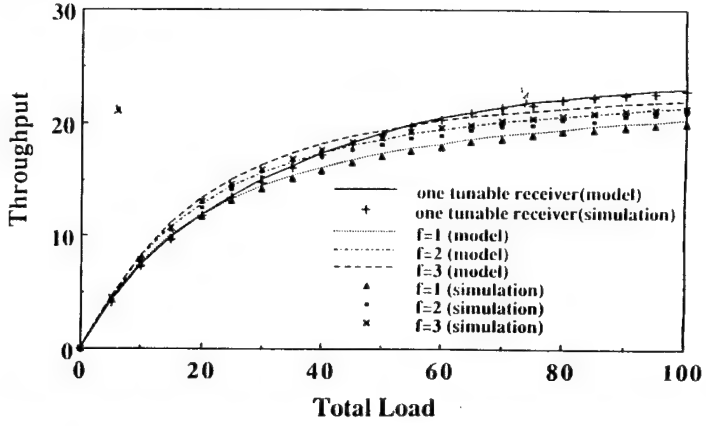


Fig. 8. Throughput versus total load $N\rho$. $N = 50$, $W = 25$, and one tunable transmitter

system ($W = N = 50$) as plotted in Fig. 9, wavelength is no longer the scarce resource and the performance of having tunability on both sides is the same as on one side only. In this case having multiple fixed receivers is always better. Note the excellent match between the results from our approximations and simulations in the figures.

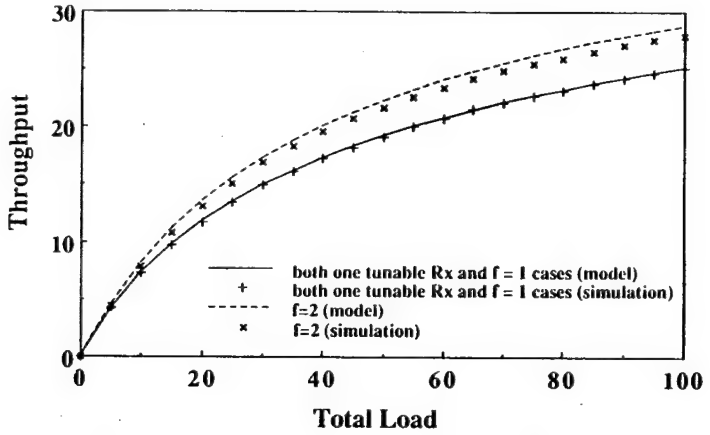


Fig. 9. Throughput versus total load $N\rho$. $N = 50$, $W = 50$, and one tunable transmitter

4. The General Traffic Case

4.1. *The Model*

Here we consider the general traffic case. We assume that node i has t_i tunable transmitters and r_i tunable receivers only. Let λ_i^* and ϕ_i^* denote the number of packets successfully transmitted and received by node i per unit time, respectively. Clearly, $S = \sum_{i=1}^{N} \lambda_i^* = \sum_{i=1}^{N} \phi_i^*$. Thus $\alpha_k^{(i)}$ can be approximated as follows:

$$\alpha_k^{(i)} = \begin{cases} 1 & 0 \le k \le t_i - 1 \\ 1 - \left(k\lambda_i^*/t_i S\right)^{t_i} & t_i \le k \le \min\left(t_i S/\lambda_i^*, W-1\right) \\ 0 & \min\left(t_i S/\lambda_i^*, W-1\right) \le k \le W-1 \end{cases} \tag{6}$$

The quantity $(k\lambda_i^*/S)$ is the average number of busy transmitters of node i, given that the system is in state k. The $(k\lambda_i^*/t_i S)$ equals the probability that any single transmitter of node i is busy given that the system is in state k. Therefore, $(k\lambda_i^*/t_i S)^{t_i}$ is approximately equal to the probability that all of node i's transmitters are busy, and one minus that gives us the $\alpha_k^{(i)}$. For those k's where the value $(k\lambda_i^*/t_i S)$ is greater than one, we set the $\alpha_k^{(i)}$ to zero. The $\beta_k^{(i)}$'s can be derived in a similar way:

$$\beta_k^{(i)} = \begin{cases} 1 & 0 \le k \le r_i - 1 \\ 1 - \left(k\phi_i^*/r_i S\right)^{r_i} & r_i \le k \le \min\left(r_i S/\phi_i^*, W-1\right) \\ 0 & \min\left(r_i S/\phi_i^*, W-1\right) \le k \le W-1 \end{cases} \tag{7}$$

Note that when the traffic is uniform, $\lambda_i^*/S = \phi_i^*/S = 1/N$, $1 \le i \le N$ by symmetry, and (6) and (7) both reduce to (5).

We now derive λ_i^*. Let $U^{(i)}$ denote the number of busy transmitters of node i in steady state with probability mass function (pmf) $u_m^{(i)} \triangleq Prob[U^{(i)} = m]$. We will approximate $U^{(i)}$ as a Markov process. Define

$$p_{k|m} \triangleq Prob[K = k \mid K \ge m] = p_k / \sum_{j=m}^{W} p_j$$

Let $\eta_m^{(i)}$ be the transition rate for $U^{(i)}$ from state m to state $m+1$. $\eta_m^{(i)}$ can be calculated as follows:

$$\eta_m^{(i)} = \lambda_i \sum_{j=1}^{N} x_{ij} \sum_{k=m}^{W-1} \beta_k^{(j)} p_{k|m}$$

The transition rate from state m to $m-1$ is just $m\mu$, the aggregrate rate at which any busy transmitter of node i will finish its transmission first. Figure 10 shows the state transition diagram. Solving this, we have

$$u_m^{(i)} = u_0^{(i)} \prod_{n=0}^{m-1} \frac{\eta_n^{(i)}}{(n+1)\mu}$$

where

$$u_0^{(i)} = \left[1 + \sum_{m=1}^{t_i} \prod_{n=0}^{m-1} \frac{\eta_n^{(i)}}{(n+1)\mu}\right]^{-1}$$

λ_i^* can be obtained from

$$\lambda_i^* = \sum_{m=0}^{t_i} m u_m^{(i)} \tag{8}$$

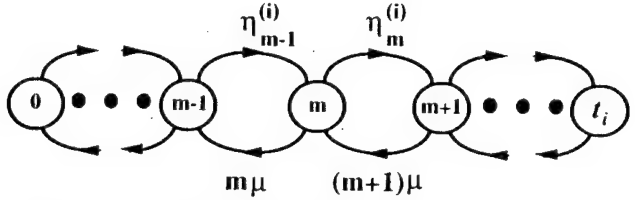


Fig. 10. State transition diagram for $U^{(i)}$

The ϕ_i^* can be derived in almost the same way. Let $V^{(i)}$ denote the number of busy receivers of node i in steady state with pmf $v_m^{(i)} \triangleq Prob[V^{(i)} = m]$. Define $\tau_m^{(i)}$ as the transition rate for $V^{(i)}$ from state m to state $m+1$. The $\tau_m^{(i)}$ can be calculated as follows:

$$\tau_m^{(i)} = \sum_{j=1}^{N} \lambda_j x_{ji} \sum_{k=m}^{W-1} \alpha_k^{(j)} p_{k|m}$$

The transition rate from state m to $m-1$ is just $m\mu$, the aggregrate rate at which any busy receiver of node i will finish its reception first. Figure 11 shows the state transition diagram. Solving this, we have

$$v_m^{(i)} = v_0^{(i)} \prod_{n=0}^{m-1} \frac{\tau_n^{(i)}}{(n+1)\mu}$$

where

$$v_0^{(i)} = \left[1 + \sum_{m=1}^{r_i} \prod_{n=0}^{m-1} \frac{\tau_n^{(i)}}{(n+1)\mu} \right]^{-1}$$

The ϕ_i^* can be obtained from

$$\phi_i^* = \sum_{m=0}^{r_i} m v_m^{(i)} \tag{9}$$

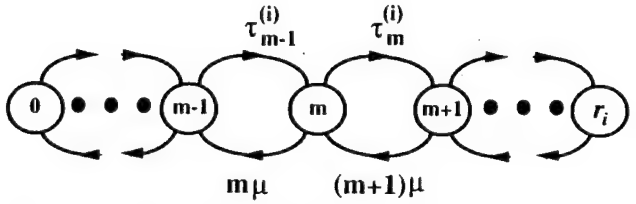


Fig. 11. State transition diagram for $V^{(i)}$

However, we do not really have the p_k's in the first place to compute those λ_i^* and ϕ_i^* because they depend on each other. In the next subsection, we propose an iterative procedure to solve for these steady state probabilities.

4.2. *An Iterative Procedure*

We define $p_k(n)$, $\lambda_i^*(n)$, $\phi_i^*(n)$, $\alpha_k^{(i)}(n)$, and $\beta_k^{(i)}(n)$ as the values obtained for these quantities at the end of the nth iteration. We start with some initial estimates $p_k(0)$, $\lambda_i^*(0)$, and $\phi_i^*(0)$. One simple initial estimate is to set $p_k(0) = 1/(W+1)$, $0 \le k \le W$, $\lambda_i^*(0) = \lambda_i$, and $\phi_i^*(0) = \phi_i$, $1 \le i \le N$. The iterative procedure is as follows:

1. Let $n = 1$.
2. Construct $\alpha_k^{(i)}(n)$ and $\beta_k^{(i)}(n)$ from $\lambda_i^*(n-1)$ and $\phi_i^*(n-1)$ using (6) and (7). Solve for $p_k(n)$ from (1), (2), and (3).
3. With $p_k(n)$, $\alpha_k^{(i)}(n)$, and $\beta_k^{(i)}(n)$, solve the Markov chains in Fig. 10 and 11 to get $\lambda_i^*(n)$ and $\phi_i^*(n)$.
4. If the difference between $p_k(n)$, $\lambda_i^*(n)$, $\phi_i^*(n)$, and $p_k(n-1)$, $\lambda_i^*(n-1)$, $\phi_i^*(n-1)$, respectively, are less than pre-specified thresholds, then stop. Otherwise, set $n = n+1$ and go to step 2.

We do not have proof of the convergence of the procedure above. However, for all the experiments presented in the next section, this procedure converges all the time, and the solutions are very close to the simulation results.

5. The Hot-Spot Traffic Case

Here we use the general model just described to study the special case of a "hot-spot" traffic pattern where a large portion of traffic is addressed to a specific node called the hot-spot node. The other $N-1$ nodes are called "plain" nodes. Without loss of generality, let node 1 be the hot-spot node. We assume all $\lambda_i = \lambda$, $1 \le i \le N$. From the generated traffic from all the nodes, a fraction of b is assumed to go to the hot-spot node, and the rest goes to the other nodes uniformly, i.e., $x_{i1} = b$, $x_{ij} = (1-b)/(N-1)$, $1 \le i \le N$, $2 \le j \le N$. Each node has one tunable transmitter and one tunable receiver except node 1, which may have more than one tunable receiver. That is, $t_i = 1$, $1 \le i \le N$, $r_1 \ge 1$, and $r_j = 1$, $2 \le j \le N$. The effect of various values of b and r_1 on the system performance is investigated below.

Figure 12 shows the relationship between the throughput and total load for the case of $N = 50$, $W = 10$, and $r_1 = 1$. We can see that as the bias b gets larger, the total throughput of the system is degraded. This is because, while the single receiver of the hot-spot node is overloaded, there is not enough traffic generated for exchange among the other nodes.

Since the receiver of the hot-spot node is now the scarce resource, we next study the effect of increasing the number of receivers at the hot-spot node. In Fig. 13 and 14 we plot the *received* throughput (i.e., ϕ_1^*) of the hot-spot node (node 1 in our example) versus the total load for the cases of $N = 50$, $W = 10$, $b = 0.2$, and $b = 0.8$, respectively. We note that, by increasing the number of receivers at the hot-spot node, its throughput can be improved. However, as the load increases, we see that the received throughput of the hot-spot node saturates at some value no matter how large a number of receivers it has. This is because when the total load is very heavy, the throughput of the system approaches W, and the received throughput of each node saturates at some value determined by the traffic imbalance. Putting in a lot more receivers at the hot-spot node will not help further increase its received throughput. To compute this saturated throughput for the hot-spot node (node 1), we let the load λ go to infinity. Define H as the number of busy receivers of node 1 in steady state with pmf $\pi_m \triangleq Prob[H = m]$, $0 \le m \le r_1$. The transition rate of H moving out of state m can be calculated as follows: We first note that as λ goes to infinity, there are W packets in transmission in the system all the time. Given $H = m$, we know that there are m packets going

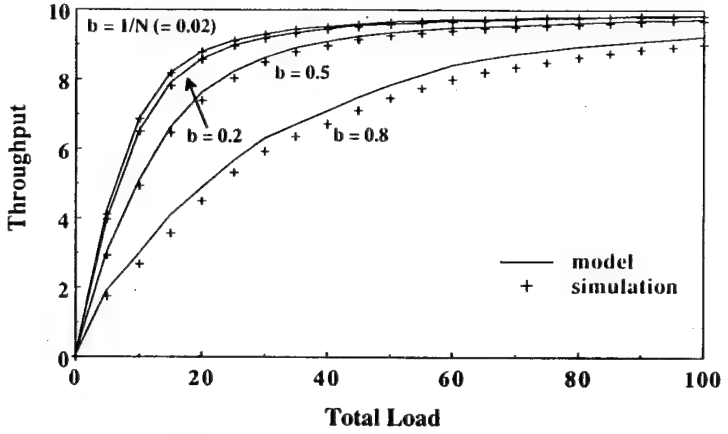


Fig. 12. Throughput versus the total load $N\rho$. $N = 50$, $W = 10$, $r_1 = 1$

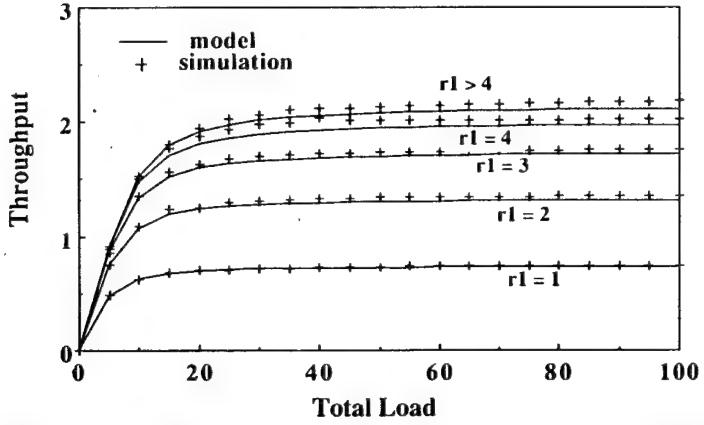


Fig. 13. Throughput of the hot-spot node versus total load $N\rho$. $N = 50$, $W = 10$, $b = 0.2$

to node 1 and $W - m$ to the others. The number of busy receivers of node 1 will increase by one when the transmission of any of the $W - m$ packets addressed to the plain nodes is finished first (with rate $(W - m)\mu$) and the wavelength just freed is immediately grabbed by a new packet addressed to node 1 (packets arrive infinitely fast since $\lambda \to \infty$), the probability of which we denote by y_m. To compute y_m, we note that, right after the transmission of any of the $W - m$ packets addressed to the plain nodes is finished, there are currently $(N-1)-(W-m)+1 = (N-W+m)$

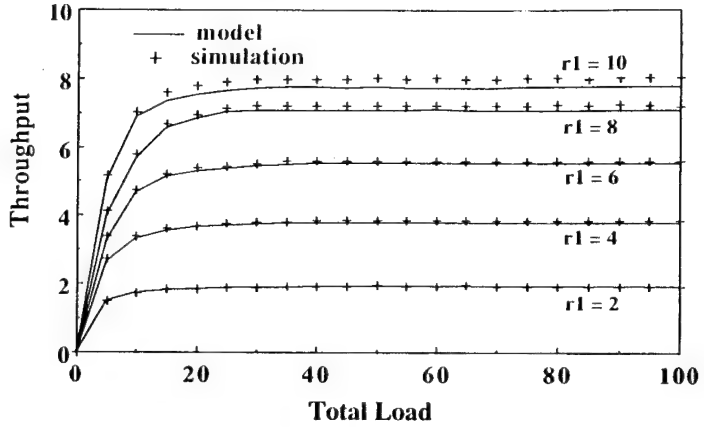


Fig. 14. Throughput of the hot-spot node versus total load $N\rho$. $N = 50$, $W = 10$, $b = 0.8$

plain nodes whose receivers are idle. The probability that the next arriving packet is addressed to the hot-spot node is equal to

$$y_m = \frac{b}{b + (N - W + m)\frac{1-b}{N-1}}$$

Therefore, the rate of H moving from m to $m+1$ equals $(W-m)\mu y_m$. On the other hand, H will decrease by one if the transmission of any of the m packets to node 1 finishes first (with rate $m\mu$) and the free wavelength is then occupied by a new packet addressed to a plain node, the probability of which is just $(1 - y_{m-1})$ because there are $(N-1)-(W-m) = (N-W+m-1)$ plain nodes whose receivers are idle. Thus, the rate of H moving from m to $m-1$ is $m\mu(1-y_{m-1})$. The state transition diagram is shown in Fig. 15. Solving this, we have

$$\pi_m = \pi_0 \binom{W}{m} \prod_{j=0}^{m-1} \frac{y_j}{1-y_j} \qquad 1 \le m \le r_1$$

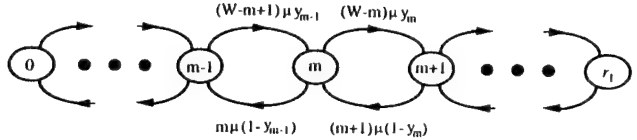


Fig. 15. State transition diagram for H

where

$$\pi_0 = \left[1 + \sum_{m=1}^{r_1} \binom{W}{m} \prod_{j=0}^{m-1} \frac{y_j}{1-y_j}\right]^{-1}$$

The real received throughput of node 1 as $\lambda \to \infty$, S_1, can be calculated from

$$S_1 = \sum_{m=0}^{r_1} m\pi_m$$

Figure 16 shows S_1 versus the number of receivers of node 1 for the case of $N = 50$ and $W = 10$. We see that, given an extremely heavy load and a large number of receivers, the hot-spot node can achieve a larger asymptotic throughput as the fraction of traffic addressed to the hot-spot node gets larger.

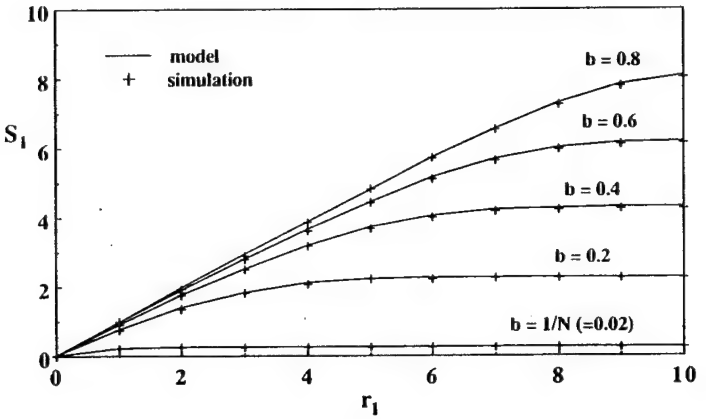


Fig. 16. The relationship between the limiting value S_1 and r_1. $N = 50$, $W = 10$

6. Conclusions

Optical fiber provides a huge amount of potential bandwidth and the bottleneck to tapping this enormous bandwidth lies at the electronic interface of the end stations. WDMA holds great promise for achieving large-scale concurrency in an optical fiber by allowing multiple communication pairs to exchange data on different channels simultaneously. In this paper we first built a model to analyze the uniform traffic case. We found that it is better to have both tunable transmitters and tunable receivers than having only one or the other tunable when the number of wavelengths is smaller than the number of nodes (which is most likely the case in the near future [14]). Also a small number of tunable transmitters and receivers at each station is enough to produce performance close to the upper bound. We then constructed a model for systems with general hardware configurations and

arbitrary traffic patterns. An iterative procedure was proposed to solve the model numerically. We used this model to study a special hot-spot traffic case. We saw that traffic imbalance could degrade the performance of the system. Adding more receivers to the hot-spot node helps improve its performance, but only to a limited extent determined by the traffic imbalance. The match between the results from our approximations and simulations was shown to be excellent.

References

[1] P. S. Henry, "High-Capacity Lightwave Local Area Networks", *IEEE Commun. Mag.* **27**, (Oct. 1989), 20–26.

[2] A. S. Acampora, "A Multichannel Multihop Local Lightwave Networks", pp. 37.5.1–37.5.9 in *Globecom '87*, Nov. 1987.

[3] M. G. Hluchyj and M. J. Karol, "ShuffleNet: An Application of Generalized Perfect Shuffles to Multihop Lightwave Networks", pp. 4B.4.1–4B.4.12 in *Infocom '88*, Mar. 1988.

[4] H. S. Stone, "Parallel Processing with the Perfect Shuffle", *IEEE Trans. Comp.* **C-20**, (Feb. 1971), 153–161.

[5] J. A. Bannister, L. Fratta and M. Gerla, "Topological Design of the Wavelength Devision Optical Network", pp. 1005–1013 in *Infocom '90*, Jun. 1990.

[6] J. Labourdette and A. S. Acampora, "Wavelength Agility in Multihop Lightwave Networks", pp. 1022–1029 in *Infocom '90*, Jun. 1990.

[7] A. Ganz and Z. Koren, "WDM Passive Star – Protocols and Performance Analysis", pp. 9A.2.1–9A.2.10 in *Infocom '91*, 1991.

[8] M.-S. Chen, N. R. Dono and R. Ramaswami, "A Media-Access Protocol for Packet-Switched Wavelength Division Multiaccess Metropolitan Area Networks", *IEEE J. Select. Areas. Commun.* **8** (6), (Aug. 1990), 1048–1057.

[9] Y. Ofek and M. Sidi, "Design and Analysis of a Hybrid Access Control to an Optical Star Using WDM", pp. 2A.3.1–2A.3.12 in *Infocom '91*, 1991.

[10] I. M. I. Habbab, M. Kavehrad and C. W. Sundberg, "Protocols for Very High-Speed Optical Fiber Local Area Networks Using a Passive Star Topology", *J. Lightwave Technol.* **LT-1** (4), (Dec. 1987), 1782–1794.

[11] R. Ramaswami and R. Pankaj, *Tunability Needed in Multi-Channel Networks: Transmitters, Receivers, or Both?*, IBM Research Report 16237 (#72046), (1990).

[12] I. Chlamtac and A. Ganz, "Design Alternatives of Asynchronous WDM Star Networks", pp. 23.4.1–23.4.6 in *ICC '89*, 1989.

[13] L. Kleinrock, *Queueing Systems, Vol. I: Theory*, John Wiley and Sons, New York, 1975.

[14] C. A. Brackett, "Dense Wavelength Division Multiplexing Networks: Principles and Applications", *IEEE J. Select. Areas. Commun.* **8** (6), (Aug. 1990), 948–964.

L K. 5

Depth-First Heuristic Search on a SIMD Machine *

Curt Powley, Chris Ferguson, and Richard E. Korf
Computer Science Department
University of California, Los Angeles
Los Angeles, Ca. 90024
(213)206-5383

July 30, 1991

Abstract

We present a parallel implementation of Iterative-Deepening-A*, a depth-first heuristic search, on the single-instruction, multiple-data (SIMD) Connection Machine. Heuristic search of an irregular tree represents a new application of SIMD machines. The main technical challenge is load balancing, and we explore three different techniques in combination. We also present a simple method for dynamically determining when to stop searching and start load balancing. We achieve an efficiency of 67%, for a speedup of 5500, with 8K processors, and an efficiency of 57%, for a speedup of 9300, with 16K processors.

*This work is supported in part by W. M. Keck Foundation grant number W880615, NSF Biological Facilities grant number BBS 87 14206, the Defense Advanced Research Projects Agency under Contract MDA 903-87-C0663, an NSF Presidential Young Investigator Award to the third author, Rockwell International, the Advanced Computing Facility, Mathematics and Computer Science Division, Argonne National Laboratory, and by Thinking Machines Corporation.

searches, or iterations. In each iteration, a branch is cut off when the $f(n)$ value of the last node on the path exceeds a cost threshold for that iteration. The threshold for the first iteration is set to the heuristic value of the initial state, and each succeeding threshold is set to the minimum f value that exceeded the previous threshold. Successive iterations continue until a goal node is chosen for expansion. Since at any point it is performing a depth-first search, IDA*'s memory requirement is only linear in the solution depth. As a result, IDA* can find optimal solutions to the Fifteen Puzzle, but some problem instances require tens of hours on current uniprocessors, motivating the use of parallel processing.

1.2 Parallel Heuristic Search

Parallel processing can dramatically reduce the time of a search algorithm. There are essentially three different approaches to parallelizing search algorithms. The first is to parallelize the tasks associated with the processing of individual nodes, such as move generation and heuristic evaluation. This is the approach taken by HITECH [5] and Deep-Thought[15], both of which use special-purpose hardware in an eight by eight array to compute chess moves in parallel. The speedup achievable in this scheme is limited, however, by the degree of parallelism available in move generation and evaluation. In addition, this approach is inherently domain-specific, and unlikely to lead to general techniques for using parallel processors to accelerate search.

A second approach is called parallel window search. This method gives each processor the entire tree to search, but divides the range of the cost bounds (windows) among the processors. Parallel window search was originated by Gerard Baudet [1] for use in searching two-player games trees [24], but has also been applied to IDA* [27][28][29], where different processors are assigned different thresholds to search. When a processor completes searching to its assigned threshold, it then searches the next unassigned threshold. Unfortunately, this approach is limited by the time to perform the goal iteration. To overcome this limitation, parallel window search can be combined with node ordering to reduce the time taken by the last iteration.

The third, and perhaps most obvious approach, is tree decomposition. While parallel window search assigns each processor the entire tree to search, tree decomposition divides the tree into subtrees, assigning different subtrees to different processors. In principle, tree decomposition allows the effective

2 Overview

We first present our basic algorithm, SIDA*, which stands for SIMD IDA*. The main idea is to divide the tree, and assign different processors to search different subtrees. By itself, this results in very poor speedup, since in an irregular tree, some processors will finish searching their subtrees long before others, and must remain idle while waiting for the others to complete. Thus, the main challenge is to perform effective load balancing in order to maintain nearly equal amounts of work on each processor and minimize idle processor overhead. Load balancing can be used in the initial distribution of nodes to processors, between successive iterations of an iterative-deepening search, and also within a particular iteration. Choosing the right times to load balance is critical to performance, and we present two different load-balance triggering strategies. We show that overall efficiency can be decomposed into the product of four different intuitive performance parameters. We then discuss the results of implementing SIDA* on a Connection Machine for the Fifteen Puzzle domain, and demonstrate a speedup of 5500 on 8K processors, and 9300 on 16K processors. Next, we present an analytic model, based on an exponential distribution of work, and analyze performance as a function of the load-balancing mechanism. We also compare our work to several other SIMD tree-search algorithms. Finally, we discuss further work and our conclusions.

IDA* is merely one example of a depth-first search of an irregular tree. Other examples include two-player alpha-beta minimax search, backtracking for constraint-satisfaction problems, and depth-first branch-and-bound for combinatorial optimization problems. The techniques presented here are largely independent of the domain and particular search algorithm, and hence should be applicable to other depth-first searches as well.

3 Basic algorithm

The basic SIDA* algorithm consists of an initial distribution of frontier nodes to processors, followed by a series of iterations of IDA*. In each iteration, each processor independently performs a depth-first search of the subtree below its frontier node. An iteration is not complete until all processors have completed their searches. During an iteration, when enough processors

3.2 SIMD Depth-First Search

Once the initial distribution is completed, each processor conducts a depth-first search of its assigned frontier node. Although processors independently perform their searches on their own subtrees, they all use the same global cost threshold. Collectively, the group of processors performs an iteration of IDA* on the entire tree, with each processor performing its iteration on its own subtree. When all subtrees have been explored, the global threshold is increased for the next iteration of IDA*.

Processors use a stack to represent the current search path in the tree. If we think of the root of the tree being at the 'top', then generating a state corresponds to moving down the stack, and returning to a previously generated state in a shallower part of the tree corresponds to 'backing up' the stack.

The stack can either contain the sequence of moves associated with the path, or can contain the sequence of states created by making moves on the path. The first approach requires only storing the current state in the search. Generating a new node involves just moving one tile in this global state. In backing up, it is necessary to 'undo' the previous move by moving the tile back to its previous location. This approach is analogous to what a person would do in solving a Fifteen Puzzle by hand.

The second approach is to make a separate copy of each newly generated state, with the appropriate modification, and place this on the stack. This approach has the advantage of not requiring a move to be undone when backing up the stack. It has the disadvantage of the overhead required to copy each state. For most problems, it is faster to record an incremental change to the state, than to copy the entire state, and hence we adopt the former approach.

Unfortunately, the trees generated by almost all heuristic searches have irregular branching factors and depths. This means that during the search phase, each processor will have a different size subtree to search, and once a processor completes its search, it must wait for all other processors to complete, before beginning the next iteration. This results in tremendous idle-processor overhead, since the variation in the sizes of the individual subtrees is enormous in practice. Thus, load balancing is necessary to effectively use a large number of processors, and is a central focus of this research.

set of nodes of minimum f value are expanded in parallel. The threshold for the first iteration of the search phase is set to the minimum f value of all the frontier nodes at the end of the distribution phase. Those processors assigned nodes whose f value is greater than the minimum do not initially participate in the first search phase, and only become active after the first load balance.

Since the distribution phase is done only once during the start of the search, regardless of how accurately the load is balanced during this phase, the loads will become unbalanced as the search progresses through multiple iterations.

4.2 Load Balancing Between Iterations

After each iteration of the search phase is completed, there is another opportunity for load balancing, based on information collected during the previous iteration. In particular, the actual number of nodes expanded below each frontier node during the last iteration, the *load* of that node, is a lower bound on the number that must be expanded on the next iteration. Furthermore, in the absence of any further information, we expect the relative loads of different frontier nodes to remain reasonably stable from one iteration to the next. In other words, a node with a relatively heavy load during one iteration is likely to have a relatively heavy load during the next iteration as well. Given this assumption, we would like to reallocate processors to frontier nodes so that lightly loaded nodes lose their processors to more heavily loaded ones.

The challenge is to accomplish this reallocation while maintaining the one-to-one correspondence between frontier nodes and processors. Consider a frontier node that was relatively lightly loaded on the last iteration, and whose brothers were also lightly loaded. We can *contract* this node by discarding the children, and making the parent a frontier node. One of the processors assigned to the children is then assigned to the parent, and the remaining processors are free to be assigned elsewhere. Node contraction was introduced by Chakrabarti et al [2], to save memory in a serial version of A*, and also used by Evett et al [7] in a SIMD version of A*.

Now consider a node that was relatively heavily loaded on the last iteration. It is expanded, generating its children. The parent now becomes an interior node, and the processor assigned to the parent is reassigned to one of the children. The remaining children are assigned processors from the

processor writes its processor ID to a location associated with its sequence number in the list, which is then read by the active processor with the same numbering in its list. As a result, each active processor obtains the ID of a unique free processor to send work to. The processors are arranged in a hypercube communication network, and the time to transfer work is a function of how many hops through the network the work must travel. The rendezvous allocation method is global in that there is no consideration of matching processors that are logically close in the network. An alternative approach is to consider locality in the hope of reducing communication costs [3].

When the number of processors with work to share exceeds the number of free processors, priority is given to processors transferring nodes with lower f values. The reason is that nodes with lower f values are expected to have larger subtrees associated with them.

Work that is shared within iterations represents only a temporary reassignment of nodes to processors, and does not change the allocation of frontier nodes to processors that was in effect at the beginning of the iteration.

After the load balancing is complete, the search phase resumes, until enough processors become idle again to justify another phase of load balancing. Thus, each iteration consists of an alternating sequence of search and load balancing phases.

5 When To Load Balance

An important problem is determining when to trigger load balancing. If load balancing is performed too frequently, then a large load balancing cost is incurred. If load balancing is not performed frequently enough, then low utilization will result. We discusses two solutions to this problem.

5.1 Constant Triggering

The simplest approach is to set a constant load-balance *trigger* X, between 0 and 1. After the initial work distribution, all P processors run until the number of processors remaining active drops to $X * P$, meaning $(1 - X) * P$ processors have become idle. When this trigger X is reached, a load balancing phase takes place in which some of the work on the busy processors

respect to t, and setting it to 0.

$$\frac{d}{dt}\frac{W(t)}{t+L} = \frac{W(t)}{-(t+L)^2} + \frac{W'(t)}{t+L} = 0 \Rightarrow \frac{W(t)}{t+L} = W'(t)$$

$W'(t)$, the derivative of $W(t)$, is the instantaneous rate at which work is being done by the system at time t, which is just the number of active processors at time t, or $A(t)$. Thus, the average rate of work over the current search/load balance cycle is maximized by load balancing when $W(t)/(t+L) = A(t)$. Since $A(t)$ changes discretely, load balancing should be performed as soon as $W(t)/(t+L) \geq A(t)$.

This dynamic triggering equation is quite general, and should apply to any problem where work and load balancing must be performed in distinct phases. By greedily optimizing the rate of work over each search/load balancing cycle, we hope to choose nearly optimal times for load balancing. Since the variables in the equation are easily computed by the program, this trigger is straightforward to implement. Most importantly however, this triggering mechanism automatically adjusts itself to any size problem, and to different stages within a single problem.

Figure 1 shows actual data for the way in which the number of active processors varies over time, using this dynamic trigger for an iteration of the Fifteen puzzle in which 94 million node generations were performed. The time to do load balancing is not shown. The area under the curve represents the total work done. Note that as the iteration progresses, the load-balancing trigger, which is the bottom envelope of the curve, decreases as expected. When there is less work remaining, the number of active processors drops off faster, and the trigger waits for sufficient work to be done before investing in another load balance. Also note that towards the end of the iteration, the final load balances do not reactivate all the processors. The reason is that there are not enough nodes on the tops of the stacks of the active processors to supply all the idle processors.

6 Measures of Performance

Before describing our experimental results, we describe our speedup and efficiency measures, and factor efficiency into four different components. These

measures apply to any SIMD applications that require load-balancing. Performance can be measured over any desired interval, such as the distribution phase, during a single iteration, over an entire problem, or over a group of problems. Hence, in the following discussion the term 'problem' can refer to part of a problem, an entire problem, or multiple problems.

6.1 Speedup and Efficiency

Our primary measure of performance is *speedup*, S, which is the time that would be required by the most efficient version of serial IDA* running on one processor, T_{IDA}, divided by the time required by our parallel algorithm to solve a problem using P processors, T. To save years of computation, we estimated the sequential time by running an efficient IDA* program on a single Connection Machine processor, for an entire iteration of a single problem. The work done, W_{IDA}, measured in node generations, divided by the time, T_{IDA}, gives the IDA* work rate, W'_{IDA}. The 'prime' in W'_{IDA} reflects that it is the rate of work per unit time.

Efficiency, E, is simply speedup divided by the number of processors, P. Since we are interested in knowing what factors contribute to overall efficiency, we divide efficiency into four components: raw speed ratio, fraction of time working, utilization, and work ratio. Besides providing valuable information, calculation of these factors provides a redundant check on the correctness of the efficiency calculation. We discuss each of these four factors in turn, then show that their product equals efficiency.

6.2 Raw Speed Ratio

The raw speed ratio reflects the fact that a single active processor in the parallel algorithm, SIDA*, performs work at a slower rate than a single CM processor executing serial IDA*. The main reason for this is the overhead of conditional statements on a SIMD machine. Since all processors must execute the same instruction, to execute a conditional, all processors choosing the first branch execute it, while the remaining processors are idle, and then those choosing the second branch execute it, while the first group is idle. Thus, the time to execute a conditional statement on a SIMD machine is the sum of the times to execute each branch, rather than a weighted average, as on a serial or MIMD machine.

6.5 Work Ratio

One final factor that affects performance is that the total work done by the serial and parallel algorithms may be different. On iterations prior to the goal iteration, both algorithms generate approximately the same number of nodes. The numbers are not exactly equal because the serial algorithm must start each of it its depth-first searches from the root node, whereas in the parallel algorithm each processor starts from a frontier node. This difference is insignificant in all but the easiest problems.

The final iteration, however, terminates as soon as a goal node is chosen for expansion. Depending on the order in which nodes are generated, and the location of the goal nodes, either the serial or the parallel algorithm may explore more total nodes on the final iteration. We will argue in section 7.2 that for this application, this effect should not be included in the efficiency, but is important in other applications.

The *work ratio*, N, is defined as the total work done by the serial algorithm, W_{IDA}, divided by the total work done by the parallel algorithm, W.

$$N = \text{Work Ratio} = \frac{W_{IDA}}{W}$$

6.6 Effect of Initial Distribution

The initial distribution of work to processors is a source of inefficiency which is indirectly included in several of the above factors. First, the raw speed ratio is lower during distribution because node generations require communication across processors to transmit states. Second, utilization is lower during distribution because it is impossible to obtain 100% utilization when the machine is being filled with work.

7 Empirical Results

We implemented SIDA* with the dynamic trigger of section 5.2 on a 16K processor Connection Machine 2 (CM2) [14], and ran all 100 Fifteen Puzzle problem instances from [18]. The speedups tend to be smaller on the easier problems, since they can't make full use of 16K processors, and the time of the distribution phase becomes significant. In computing the parameters derived in section 6, we treat all 100 problems as if they were a single large problem. This has the effect of weighting the difficult problems more heavily than the easy ones, which is appropriate since most of the time is spent solving the harder problems.

7.1 Overall Results

Using this approach, the speedup over all 100 problems on 16K processors was about 7800, which corresponds to an efficiency of almost 48%. A total of $W_{IDA} = 36$ billion nodes were generated by serial IDA*[18]. Dividing this by the node generation rate of IDA* on one CM processor, $W_{IDA} = 118$ nodes per second, gives an estimated total time for IDA* of $T_{IDA} = 84,746$ hours, or 9.7 years! The total time taken by SIDA* to do all 100 problems using 16K processors was $T = 10.8$ hours. The speedup is therefore $S = T_{IDA}/T = 7800$.

The average and median times to solve a single problem instance were 6.5 and 2.1 minutes, respectively, while the average and median number of node generations were 430,780,150 and 59,512,527, respectively.

The overall efficiency is the product of a raw speed ratio of .772, fraction of time working of .879, processor utilization of .839, and work ratio of .836.

The initial distribution phase requires about 33 seconds with 16K processors, and its effect on these factors is insignificant on the hardest problems, since it occurs once for each problem instance, and only accounts for a tiny fraction of the total node generations.

The .879 fraction of time working indicates that 12% of the total time was spent load balancing. 1.7% of the total time was spent on load balancing between iterations, taking about 6.7 seconds per iteration. This overhead is negligible since it consists of a small constant amount of work per iteration, and the number of iterations grows linearly with problem difficulty, while node generations grow exponentially. The rest of the load balancing time is

among the virtual processors. Therefore, regardless of whether the observed value of the work ratio is due to noise, or is consistently greater than one, the work ratio should be discounted in the efficiency measure, by setting it to one. If we set the work ratio to one, then the efficiency becomes almost 57% on 16K processors, which is the product of the remaining three factors. This is a more accurate measure of the performance of SIDA*, and thus we refer to the overall efficiency as 57%, with a corresponding speedup of over 9300. Note that in other applications, such as alpha-beta search, any parallel algorithm must search *more* nodes than the serial algorithm, and hence in these applications the work ratio must be taken into account.

Our speedup is on the order of four orders of magnitude with 16K processors. This is relative to IDA* running on one Connection Machine processor, which is about three orders of magnitude slower than current workstations, yielding a performance improvement relative to a high-performance workstation of about an order of magnitude.

7.3 Scalability

Next, we consider how our results scale as the number of processors increases. Ideally, efficiency would remain constant as the number of processors increases. In practice, however, for a given problem, as the machine gets larger, efficiency decreases somewhat because there is not enough work to utilize all the processors. Conversely, for a given size machine, as a problem gets harder, utilization increases, a smaller percentage of time is spent load balancing, and efficiency increases.

To illustrate this, we picked a sample of problems of varying difficulty. We ordered the problems of [18] by increasing nodes generated, and picked problems 1, 25, 50, 70, and 90 in that ordering. These corresponds to problem numbers 79, 81, 41, 98, and 63, respectively, of [18]. We solved each problem using from 2K to 16K processors in 2K increments, and plotted speedup versus number of processors for each problem in figure 2.

In general, the harder the problem, the greater the speedup for a given number of processors. This implies that larger machines will be more cost effective on harder problems than on easier problems, as expected. The Connection Machine, for example, is currently available with up to 64K processors.

We also ran all 100 problems on 8K processors. Counting them all as

one task, efficiency increased from 57% to 67% when going from 16K to 8K processors, based on a raw speed ratio of .846, a fraction of time searching of .936, and a utilization of .848. The work ratio, which is not included in the speedup, was .709. On the hardest problem, efficiency was 74% on either 8K or 16K processors. Of course, if a problem is too hard, even high efficiency may not be sufficient if the elapsed time is unacceptably long.

8 An Analytic Model

In order to evaluate these results and compare various load balancing strategies, an analytic model for how work is distributed and shared during an iteration of a typical parallel tree search algorithm is presented. Initially, each of P processors has an amount of work to do which is independently chosen from an exponential distribution function with a mean of M seconds: $e^{-x/M}$. The exponential is chosen as the work distribution because it is analytically tractable, thus allowing us to generate the optimal load balancing algorithm and compare its performance with that of other load balancing schemes. Work is allowed to progress, with processors that finish their work becoming idle, until a load balancing phase is begun. The load balancing phase takes L seconds, after which all processors are reactivated. When load balancing occurs with less than half the processors are busy, it will take more than one communication to reactivate all the processors, since a single processor can only send work to one idle processor at any given time. Therefore, L actually goes up when fewer processors are active. This is not taken into account in our model, but will only effect the tail end of most problems, since load balancing is generally triggered with more than half the processors still active (see Figure 1).

After load balancing, the amount of work on each of the P processors again comes from an exponential distribution. However, the mean of the new distribution must be chosen to reflect the fact that there is now less total work remaining. The memoryless property of the exponential makes this particularly convenient. If the amount of work on a processor comes from an exponential distribution, then the expected completion time of the processor is independent of how long the processor has been running. Initially, the P processors each have an expected amount of work of M seconds, resulting in an expected total amount of work of $M * P$. Load balancing begins with

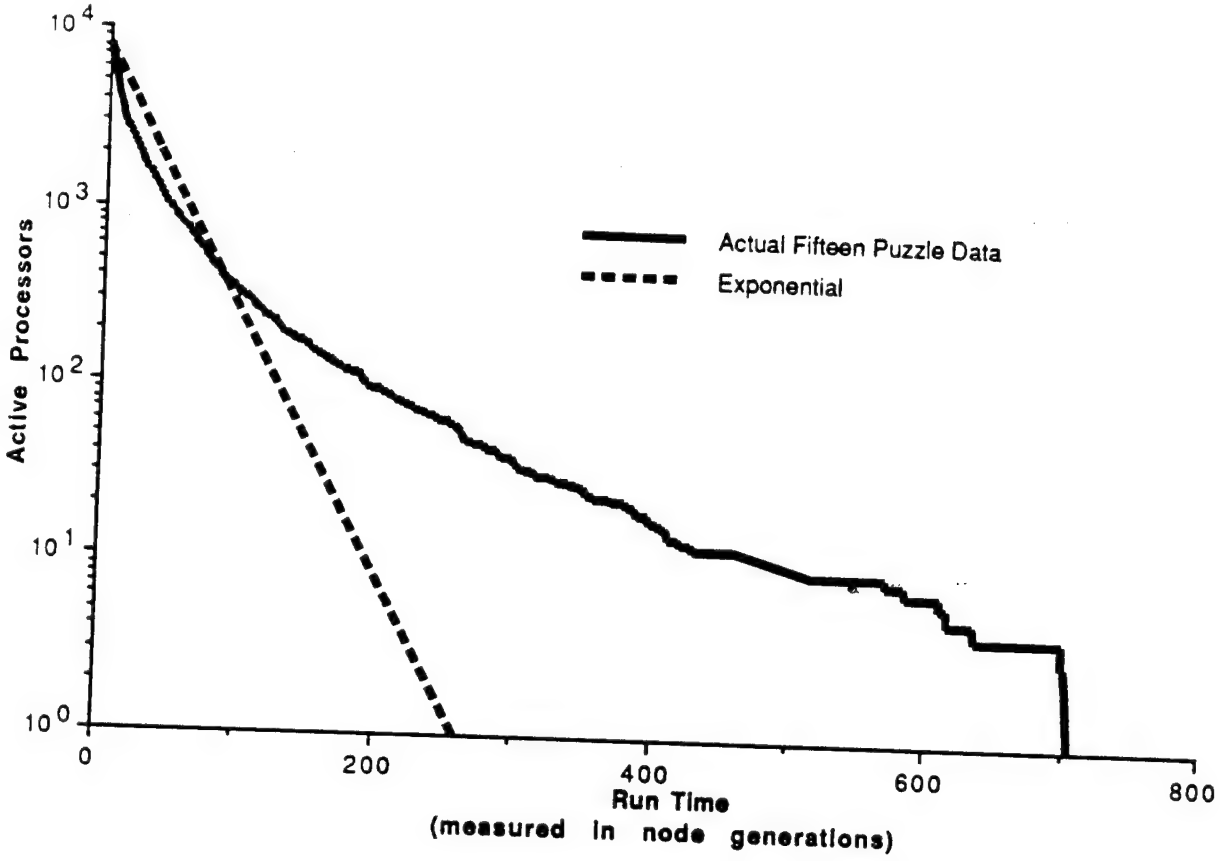


Figure 3: Actual vs. Exponential Work Distribution

load balances performed. If B is 0, the run time is approximately $M * \log(P)$ [17].

To find the minimum run time if exactly one load balance is used, we must compute the optimal trigger X at which to perform the only load balance. For $B = 1$, the expected run time, not counting the load balance, if we trigger when $X * P$ processors remain busy, is the time until the first load balance plus the completion time. The time until the first load balance will be the time for the completion of the first $P * (1 - X)$ of P exponentials with mean M, which is approximately $M * \log(1/X)$ [17]. Since the work on each processor is reduced to $M * X$ after load balancing, the completion time is approximately $M * X * \log(P)$. This yields a total search time, not counting load balancing, of

$$M * \log(1/X) + M * X * \log(P)$$

To find the optimal trigger X in the case where a single load balance is used, we must minimize the value of this equation for X. To do this we take the derivative with respect to X and set it to 0:

$$M/X + M * \log(P) = 0 \Rightarrow X = 1/\log(P)$$

Plugging this back into the equation for time, we find that the minimum run time for 1 load balance is :

$$M * \log\log(P) + M * \log(P)/\log(P) =$$

$$M * \log\log(P) + M = M * (\log\log(P) + 1)$$

Now let $M * Z_B(P)$ be the run time not counting load balances for B optimally performed load balances, running on P processors, each with an expected amount of work of M seconds. We will show by induction on B that $Z_B(P)$ is independent of M. The base cases are $Z_0(P) = \log(P)$ and $Z_1(P) = \log\log(P) + 1$. The amount of time taken if $i + 1$ optimal load balances are performed is the time to reach the first optimal trigger X, $M * \log(1/X)$, plus the completion time for i optimally performed load balances on a work distribution with a mean of $X * M$ seconds. Using the assumption that Z is independent of the mean of the work distribution, the second term is $X * M * Z_i(P)$. The total time taken is thus

efficiency as halving the number of processors, since the amount of work per processor doubles in either case. For small problems, which result in low efficiency, the amount of work per processor can be increased by reducing the number of processors. Low efficiency tells us that the problem is too small for the number of processors used, and reducing the number of processors may not greatly affect the time taken by the algorithm because of the increased efficiency.

The three best approaches all perform very well, however, both the optimal load balancing formula and the optimal constant triggering formula rely on the fact that the distribution of work associated with each processor is drawn from an exponential distribution with a known mean. The dynamic triggering formula, however, does not make this assumption, and thus is more general.

Figure 5 compares a line representing the simulated results for the model of dynamic load balancing (sections 8 and 5.2) with data points representing the efficiency of SIDA* with 16K processors on all 100 Fifteen Puzzle problem instances.

The main reason that the results do not match the model very well is that the model makes the optimistic assumption that the workload comes from an exponential distribution. Furthermore, this distribution may change over the course of a solution as load balancing is performed, thus decaying even further from exponential. Another reason is that the model computes efficiencies for a single iteration as opposed to the multiple iterations of an iterative-deepening search.

9 Related Work

Related work on SIMD search includes our own preliminary work, a best-first algorithm based on A*, a another depth-first algorithm based on IDA*, and a brute-force depth-first search.

Preliminary results of our current effort appeared in [25] and [26], which describes the overall structure of our algorithm, together with an implementation of the distribution and search phases, including load balancing during the distribution phase.

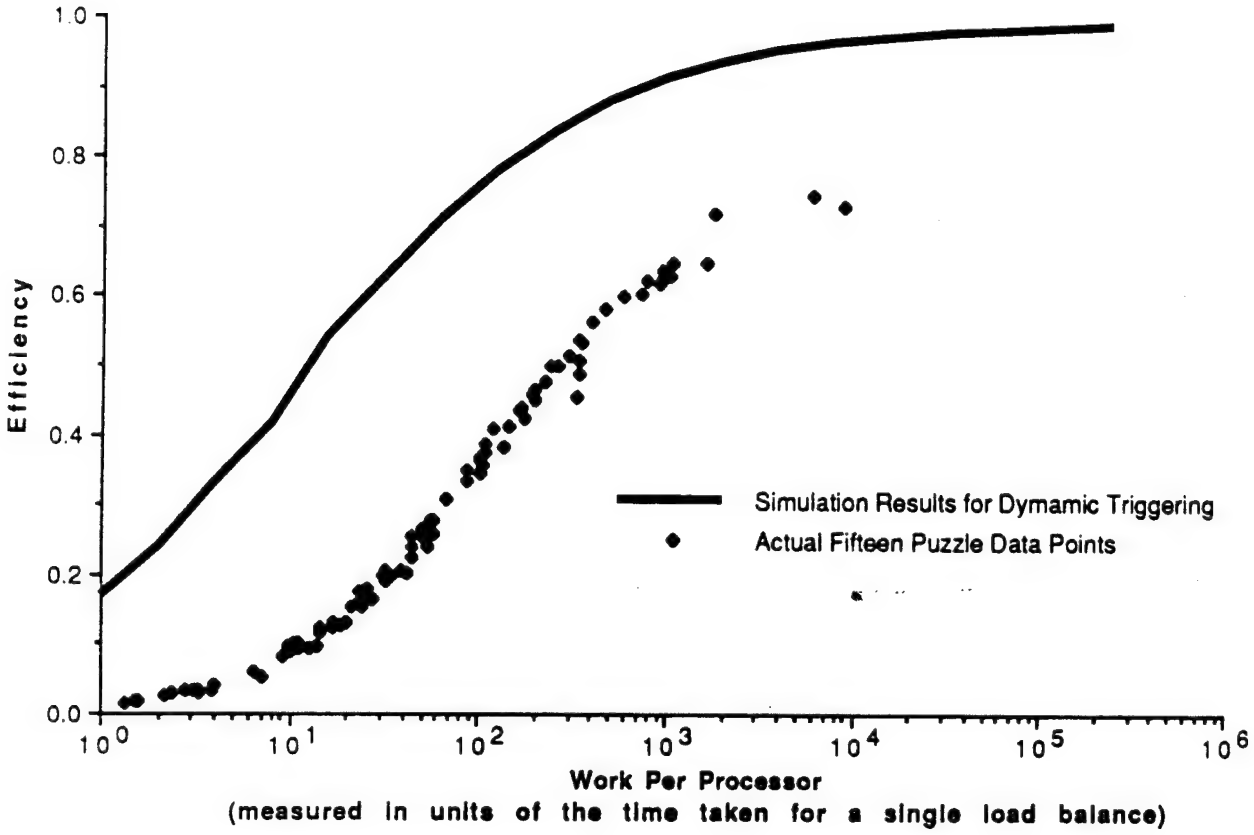


Figure 5: Theoretical and Empirical Results for Dynamic Trigger

9.2 IDPS

IDPS (Iterative-Deepening Parallel Search) [20] is a SIMD search algorithm based on IDA*, also using the Fifteen Puzzle domain, and developed concurrently with SIDA*. Like SIDA*, it uses an initial heuristic distribution of nodes to processors, followed by alternating phases of search and load balancing.

There are several differences between IDPS and SIDA*. SIDA* generates nodes by making changes to a single copy of the state, and maintains the structure of the search tree, whereas IDPS generates new nodes by copying the entire state, and discards a parent node after all its children have been generated. It keeps track of the solution by storing the moves made with each child node. In addition, IDPS passes multiple independent nodes to a single processor during load balancing, while SIDA* only passes single nodes, or children of the same parent. Finally, IDPS does not perform between-iteration load balancing, and performs within-iteration load balancing as soon as any processor becomes idle.

Mahanti and Daniels [20] report an overall efficiency of 76% on 16K processors, and 92% for 8K processors, based on the same 100 problems from [18]. These values are not directly comparable to our corresponding efficiencies of 57% and 67%, however, for several reasons. The first is that they include the work ratio in their efficiency measure. In fact, they achieve an average superlinear speedup (efficiency greater than 100%) for problems larger than 20 million node generations, which requires a work ratio greater than one. The second difference is that our speed is based on a serial IDA* speed of 118 nodes per second, whereas the speedup of IDPS is based on a serial IDA* speed of 59 nodes per second, on the same machine[21]. However, some of the optimizations in our IDA* code may apply to IDPS as well, without effecting its efficiency. While we don't have timing data on IDPS for 16K processors, the two programs took roughly the same amount of time (21.66 hours for SIDA* vs. 22.56 hours for IDPS) to solve all 100 problem instances on 8K processors, despite the fact that, due to a low work ratio, SIDA* generated over 30% more nodes[21].

pha and beta bounds that are produced by the search of one subtree effect the efficiency of searching later subtrees. As a result, these bounds must be propagated throughout the tree, and a parallel algorithm will do more work than a serial algorithm.

Another example in this class of algorithms is depth-first branch-and-bound for combinatorial optimization problems, such as the traveling salesman problem. Since SIDA* maintains the internal structure of the tree, it should be extensible to branch-and-bound algorithms as well.

11 Conclusions

SIDA* achieves an efficiency of 67%, for a speedup of over 5500 on 8K processors, and 57%, for a speedup of over 9300 with 16K processors on the Fifteen Puzzle. This demonstrates that SIMD machines can be effectively used to search irregular, dynamically generated trees. As SIMD machines increase in size and speed relative to uniprocessors, their use for heuristic search applications should become increasingly cost-effective.

The more general approach of alternating phases of work followed by load balancing should apply to other irregular computations on a SIMD machine, such as Monte-Carlo methods[12]. We present a very general technique for deciding when to halt work and begin load balancing. This dynamic load-balancing trigger is simple to compute, makes locally greedy decisions, and achieved relatively high overall utilization with modest overhead in our experiments.

Acknowledgements

Rob Collins provided enormous help on the use of the Connection Machine. We also thank Ambuj Mahanti for many discussions on this research topic.

References

[1] Gerard Baudet, "The Design and Analysis of Algorithms for Asynchronous Multiprocessors", Ph.D. dissertation, Computer Science Department, Carnegie-Mellon Univ., Pittsburgh, Pa., April 1978.

Proceedings of the Unstructured Scientific Computation on Multiprocessors Conference, P. Mehrotra, S. Sulte, and R. Voight (Eds.), MIT Press, 1991.

[13] P. E. Hart, N.J. Nilsson, and B. Raphael, "A Formal Basis For The Heuristic Determination of Minimum Cost Paths", *IEEE Trans. Systems Sci. Cybernetics*, 4(2), 1968, pp. 100-107.

[14] W. Daniel Hillis, *The Connection Machine*, MIT Press, 1986.

[15] Hsu, F.-H., T. Anantharaman, M. Campbell, and A. Nowatzyk, A grandmaster chess machine, *Scientific American*, Vol. 263, No. 4, Oct. 1990, pp. 44-50.

[16] Karp, R.M., "Reducibility among combinatorial problems", in R.E. Miller and J.W. Thatcher (eds.), *Complexity of Computer Computations*, Plenum Press, New York, 1972, pp. 85-103.

[17] Leonard Kleinrock, *Queueing Systems. Volume 1: Theory*, John Wiley, 1975.

[18] Richard E. Korf, "Depth-First Iterative-Deepening: An Optimal Admissible Tree Search", *Artificial Intelligence*, Vol. 25, 1985, pp. 97-109.

[19] Vipin Kumar and V. Nageshwara Rao, "Parallel Depth-First Search, Part II: Analysis", *International Journal of Parallel Programming*, Vol. 16(6), 1987, pp. 501-519.

[20] Ambuj Mahanti and Charles J. Daniels, "SIMD Parallel Heuristic Search", Technical Report Nos. UMIACS-TR-91-41, CS-TR-2633, Computer Science Department, University of Maryland, College Park, Maryland, May, 1991.

[21] Ambuj Mahanti, personal communication, June 1991.

[22] Giovanni Manzini and Marco Somalvico, "Probabilistic Performance Analysis of heuristic Search Using Parallel Hash Tables", *Proceedings of the International Symposium on Artificial Intelligence and Mathematics*, Fort Lauderdale, Florida, January, 1990.

[32] V. Nageshwara Rao and Vipin Kumar, "Superlinear Speedup in Ordered Depth-First Search", *Proceedings of the 6th Distributed Memory Computing Conference (DMCC6)*, May 1991.

[33] Ratner, D., and M. Warmuth, "Finding a shortest solution for the NxN extension of the 15-Puzzle is intractable", *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI86)*, Philadelphia, Pa., 1986, pp. 168-172.

58	100	67880056	51065359	2.1	0.495	0.652	0.644	1.329	0.208
59	99	83477694	60664396	2.3	0.536	0.634	0.663	1.376	0.225
60	44	95733125	58844809	2.1	0.541	0.652	0.681	1.627	0.240
61	25	100734844	113108217	2.9	0.637	0.706	0.747	0.891	0.336
62	84	106074303	156836306	3.3	0.700	0.748	0.785	0.676	0.411
63	69	109562359	59512527	2.5	0.470	0.660	0.658	1.841	0.204
64	29	117076111	138153686	3.4	0.638	0.728	0.743	0.847	0.345
65	76	126638417	337475877	5.8	0.762	0.818	0.802	0.375	0.501
66	80	132945856	135550244	3.3	0.615	0.783	0.742	0.981	0.358
67	11	150346072	43724410	1.9	0.498	0.633	0.638	3.438	0.201
68	70	151042571	190221310	3.9	0.686	0.767	0.790	0.794	0.416
69	89	166571097	78899335	2.6	0.528	0.693	0.706	2.111	0.258
70	98	183526883	223791797	4.4	0.706	0.778	0.797	0.820	0.438
71	10	198758703	231574183	4.5	0.712	0.783	0.789	0.858	0.441
72	54	220374385	222102072	4.4	0.707	0.775	0.793	0.992	0.435
73	26	226668645	264329624	5.0	0.749	0.767	0.782	0.858	0.449
74	67	252783878	387048354	7.3	0.684	0.895	0.744	0.653	0.455
75	21	257064810	228854282	4.3	0.720	0.811	0.787	1.123	0.459
76	64	260054152	146886788	3.3	0.689	0.731	0.767	1.770	0.386
77	1	276361933	252918505	4.7	0.742	0.789	0.794	1.093	0.465
78	37	280078791	420269004	7.0	0.778	0.813	0.815	0.666	0.517
79	27	306123421	222661611	4.5	0.699	0.764	0.792	1.375	0.423
80	52	377141881	492677454	7.5	0.792	0.859	0.830	0.765	0.565
81	7	387138094	318081544	5.5	0.756	0.824	0.805	1.217	0.501
82	53	465225698	1302507835	18.0	0.815	0.911	0.837	0.357	0.622
83	33	480637867	314891988	5.7	0.755	0.790	0.803	1.526	0.479
84	15	543598067	491131113	7.8	0.782	0.835	0.826	1.107	0.539
85	3	565994203	439740204	7.4	0.781	0.827	0.786	1.287	0.508
86	91	602886858	791090780	11.3	0.818	0.882	0.835	0.762	0.602
87	17	607399560	467618099	7.5	0.764	0.843	0.827	1.299	0.533
88	32	661041936	168889240	3.8	0.691	0.744	0.749	3.914	0.385
89	22	750746755	633205198	9.3	0.802	0.876	0.831	1.186	0.583
90	63	995472712	455660892	8.0	0.786	0.796	0.782	2.185	0.490
91	72	1031641140	1395384280	19.1	0.824	0.908	0.842	0.739	0.631
92	92	1101072541	976975496	13.9	0.828	0.880	0.829	1.127	0.604
93	56	1199487996	1188535843	16.0	0.826	0.905	0.853	1.009	0.638
94	59	1207520464	1198818106	16.4	0.835	0.897	0.837	1.007	0.628
95	14	1369596778	1055837119	14.6	0.824	0.897	0.842	1.297	0.623
96	49	1809933698	2259207595	27.0	0.847	0.945	0.899	0.801	0.720
97	66	1957191378	1396015331	18.4	0.829	0.919	0.854	1.402	0.650
98	60	3337690331	2284080022	30.3	0.835	0.932	0.834	1.461	0.649
99	82	5506801123	7648036961	88.0	0.859	0.977	0.890	0.720	0.747
100	88	6009130748	12003614428	141.0	0.850	0.983	0.876	0.501	0.731
TOTALS		35991891900	43078015000	649.4	0.772	0.879	0.839	0.836	0.569

dn	pn	W(IDA)	W	T	R	F	U	N	E/N

Key:
dn. problem number ordered by difficulty of [Kor85]
pn. problem number of [Kor85].
W(IDA). nodes generated by serial IDA*.
W nodes generated by SIDA*.
T elapsed time of SIDA*.
R raw speed ratio.
F fraction of time working.
U utilization.
N work ratio, ratio of serial to parallel nodes generated.
E/N efficiency without work ratio, E/N = R*F*U.
(totals count all 100 problem instances as one large problem)